21615

8080 8085 Software Design

Book 2

Christopher A. Titus, David G. Larsen,
and Jonathan A. Titus

BLACKSBURG CONTINUING EDUCATION SERIES™
edited by Titus, Larsen & Titus

The Blacksburg Continuing Education™ Series

The Blacksburg Continuing Education Series™ of books provide a Laboratory—or experiment-oriented approach to electronic topics. Present and forthcoming titles in this series include:

- DBUG: An 8080 Interpretive Debugger
- Design of Active Filters, With Experiments
- Design of Op-Amp Circuits, With Experiments
- Design of Phase-Locked Loop Circuits, With Experiments
- Design of Transistor Circuits, With Experiments
- Design of VMOS Circuits, With Experiments
- The 8080A Bugbook®: Microcomputer Interfacing and Programming
- 8080/8085 Software Design (2 Volumes)
- 555 Timer Applications Sourcebook, With Experiments
- Guide to CMOS Design Basics: Circuits and Experiments
- Interfacing and Scientific Data Communications Experiments
- Introductory Experiments in Digital Electronics and 8080A Microcomputer Programming and Interfacing (2 Volumes)
- Logic & Memory Experiments Using TTL Integrated Circuits (2 Volumes)
- Microcomputer—Analog Converter Software and Hardware Interfacing
- Microcomputer Interfacing With the 8255 PPI Chip
- NCR Basic Electronics Course, With Experiments
- NCR Data Communications Concepts
- NCR Data Processing Concepts Course
- NCR EDP Concepts Course
- Programming and Interfacing the 6502
- 6800 Microcomputer Interfacing and Programming, With Experiments
- 6502 Software Design
- TEA: An 8080/8085 Co-Resident Editor/Assembler
- TRS-80 Interfacing
- Z-80 Microprocessor Programming & Interfacing (2 Volumes)

In most cases, these books provide both text material and experiments, which permit one to demonstrate and explore the concepts that are covered in the book. These books remain among the very few that provide step-by-step instructions concerning how to learn basic electronic concepts, wire actual circuits, test microcomputer interfaces, and program computers based on popular microprocessor chips. We have found that the books are very useful to the electronic novice who desires to join the "electronics revolution," with minimum time and effort.

Additional information about the "Blacksburg Group" is presented inside the rear cover.

Jonathan A. Titus, Christopher A. Titus, and David G. Larsen
"The Blacksburg Group"

Bug symbol trademark Nanotran, Inc., Blacksburg, VA 24060
Bugbook is a registered Trademark of E & L Instruments, Inc., Derby, CT 06418

PHILIP SALMON
P.O. Box 345
DRUMMOYNE NSW 2047
*a/d Phone: (02) 818 1111

8080/8085
Software Design
Book 2

by

**Christopher A. Titus, David G. Larsen,
and Jonathan A. Titus**

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

Copyright © 1979 by Christopher A. Titus,
David G. Larsen, and Jonathan A. Titus

FIRST EDITION
SECOND PRINTING—1980

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-21615-9
Library of Congress Catalog Card Number: 78-57207

Printed in the United States of America.

Preface

With the advent of low-cost microcomputers such as the Commodore PET and the Radio Shack TRS-80, many people are starting to use microcomputers. These two microcomputers are usually programmed in the BASIC language, although both have the capability of being programmed in assembly language. As the size and complexity of the BASIC programs for these two microcomputers increase, more and more users begin to realize some of the limitations of the BASIC interpreter that they have. In fact, some tasks are more easily performed in assembly language than in the BASIC language.

Some PET and TRS-80 users find that the BASIC interpreter has enough "power" for them, but they also find that they have to write their own assembly language subroutines so that the BASIC interpreter can communicate with the peripherals that they have interfaced to the microcomputer.

The point is, assembly language programming will be used for a long time, remaining an effective way of programming the 8080, Z-80, 6800, and 6502 microcomputers. In fact, assembly language programs can probably solve more problems than BASIC programs can! For this reason, we have added a second volume to the *8080/8085 Software Design* series.

In the last chapter of the first volume (*8080/8085 Software Design—Book 1*), we discussed a number of I/O devices and their appropriate software "drivers." The I/O devices that were used in that chapter included ASCII keyboards, 16-key hardware-encoded keyboards, scanned keyboards, latched LED displays, and multiplexed LED displays.

Although not required, we highly recommend this book. It provides a good reference for the basic 8080 instruction set, along with program examples for mathematical operations, number-base conversions, and I/O programming. There are also a number of program examples and discussions in Book 1 that we reference in this second volume.

In the first chapter in this volume, we discuss asynchronous communications using UARTs, USARTs, and the microcomputer itself to serialize data and transmit it to a peripheral device. These same devices are also used to receive serial data from a peripheral. As you will see, by using software instructions to perform the transmission and reception of data, hardware costs are minimized. We also discuss the two 8085 instructions, SIM and RIM, which are not included in the 8080 instruction set. As always, we include *electronic schematic diagrams* of the interfaces that are required for the software examples to operate properly.

The next two chapters deal with interrupts. Chapter 2 discusses interrupts in general—polled, vectored, and priority interrupts. Included in this chapter are interrupt service subroutines and the generalized structure that these subroutines must have. The interrupt applications chapter discusses real-time clocks, time-of-day clocks, and programmable clocks. Also included are programs and subroutines for interrupt-driven keyboards (hardware-encoded and scanned) and multiplexed LED displays. The last section of the chapter describes the 8214 priority interrupt control unit and discusses the software required to "drive" this integrated circuit.

The next four chapters of the book deal with data structures and how they are accessed. A short chapter defines "data structure," along with some of the common data structure terms. Following this discussion, the next chapter contains a number of subroutines that can be used to search lists or tables for specific numeric values. Subroutines are listed that can search for eight-, 16-, and 24-bit numbers. The last section of this chapter discusses alphanumeric-string searching. Examples are given where a list is searched for addresses that contain a specific ZIP code.

The chapter on sorting also deals with numeric values and alphanumeric strings. Two sorting methods are used (insertion sort and exchange sort) and best-case/worst-case times are included for some actual sorting problems.

Look-up tables are described in Chapter 7, and the program examples address the problems of generating the sine of an angle (very quickly) and punching alphanumeric characters on paper tape. A 90-node (entry) sine look-up table is used to generate the sine of any angle between 0° and 360°, in 1° increments. The sign of the sine value is also determined.

Command decoders of one type or another are probably used in 90% of all microcomputer programs. For this reason, we have included a chapter that discusses them and includes numerous program examples. The command decoder examples deal with the decoding of fixed-length and variable-length commands.

The last two chapters, Chapters 9 and 10, deal with system monitors and debuggers. Many microcomputers come from the manufacturer already programmed with one of these types of programs, including the H-8 (Heath Company), the SBC 80/10, SDK-80, and SDK-85 (Intel Corporation), the BLC 80/10 (National Semiconductor Corporation), the KIM (MOS Technology Corporation), and the AIM-65 (Rockwell International). If you are interested in writing a system monitor or debugger for the system that you are currently working with, these two chapters should give you plenty of ideas. These two chapters contain a number of programs and subroutines that you can use as a system monitor or debugger.

One goal that we had when writing this book was to provide detailed descriptive explanations of how program examples operate. We do not say, "Here it is, you determine how it works." You would learn very little if this approach was used. Instead, we *develop* programs, starting with the simplest sequence of instructions that can accomplish the task. These programs often have limitations and, if they do, we add instructions to them so that the microcomputer can better perform either a more general-purpose task, or a specific task. Therefore, we *design* the best solution to a problem, learning from previous examples.

One area that you do not have to worry about is whether you have the peripheral hardware required to execute our program and subroutine examples. We do not have hardware and software examples that feature one manufacturer's hardware. The program and subroutine examples will execute equally well on a PERTEC (MITS), Intel Corporation, National Semiconductor Corporation, Heath Company, Control Logic, or E & L Instruments 8080-based microcomputer. Of course, peripheral devices may vary from system to system, just as peripheral device addresses may vary from system to system.

In this book, we continue to use a coresident editor/assembler (TEA) that generates a byte-per-line program listing. With this type of listing, it is easier for many readers to understand *exactly* where address and data bytes of multiple-byte instructions are stored in memory. We have also learned from past experience that many people still hand-assemble programs, simply because their computers do not have enough memory to store an editor/assembler, or they cannot find an editor/assembler that will work on their system. With the program and subroutine listings that we provide, hand assembly is very easy because we have "left room" for the address and data bytes of the multiple-byte instructions. We have also included both octal *and* hexadecimal numbers in the discussions. The octal number is usually listed first, followed by the hexadecimal number in parentheses; for example, 125 (55).

The number of books in the *Blacksburg Continuing Education Series*™ is growing all the time. The books in this series that are currently available are listed on the inside of the front cover. We have books in preparation that deal with programming and interfacing for the 6800, 6502, and the Z-80. We are also in the process of writing a book about our coresident editor/assembler (TEA), complete with hexadecimal source listings. We continue to be interested in identifying and working with authors who think that their book ideas would fit into this series. If you have an idea that you are interested in working on, contact us here in Blacksburg.

We have found wide acceptance of our books in formal classes as well as by individual users worldwide. Selected books are being translated into German, Spanish, Japanese, French, Italian, Chinese, and Malaysian. If you are interested in further details concerning these translations, or in translating the books into other languages, please contact us.

Many of the concepts that are presented in this book have been incorporated into the material taught at seminars that are currently presented by Tychon, Inc., here in Blacksburg. Two courses are currently being taught: *Microprocessor Interfacing* (*628*) and *Software Design for the 8080/8085 Processors* (*690*). If you are interested in these courses, write to the Course Director, Tychon Inc., P.O. Box 242, Blacksburg, VA 24060. Courses are also provided through the Extension Division of Virginia Polytechnic Institute and State University, Blacksburg, VA 24061. Call Dr. Linda Leffel at (703) 961-5241 for further information.

CHRISTOPHER A. TITUS, DAVID G. LARSEN,
and JONATHAN A. TITUS
"The Blacksburg Group"

Contents

LIST OF PROGRAM EXAMPLES 9

CHAPTER 1

ASYNCHRONOUS SERIAL COMMUNICATIONS 13

The Hardware Method – UART Features – Serial Data Format – Hardware UART Software – Software-Based UARTs – The 8085 and UARTs – 8085-Based, Software-Based, Asynchronous Serial Receiver Software – Hardware Devices–USARTs Versus UARTs – Memory-Mapped UARTs and USARTs

CHAPTER 2

INTERRUPTS 55

Interrupt Operation – Basic Types of Interrupts – Interrupt Instructions – The Enable and Disable Interrupt Instructions – How the 8080 is Actually Interrupted – Single-Line Interrupts (Polled Interrupts) – Vectored Interrupts – Vectored and Polled Interrupts – Priority Interrupts – Hardware Priority Interrupts – The 8085 and Interrupts – Priority Interrupt Software Timing

CHAPTER 3

INTERRUPT APPLICATIONS 95

A Real-Time Clock – A Time-of-Day Clock – Interrupt-Driven Keyboards – Interrupt-Driven Scanned Keyboards – Interrupt-Driven Multiplexed Light-Emitting-Diode Displays – The 8214 Priority Interrupt Control Unit

CHAPTER 4

DATA STRUCTURES 139

Linear Lists – Sequential Allocation – Linked Allocation – Circular Lists

CHAPTER 5

SEARCHING 145

Single-Precision Lists (Eight Bit) – Double-Precision Lists (16 Bit) – Triple-Precision Lists (24 Bit) – Common Search Subroutine Characteristics – Searching ASCII Strings – Entering a Test Zip Code Into Memory – Testing the Zip Code Search Program – One Final Program Bug

CHAPTER 6

SORTING 181
Sorting Numeric Values – Sorting Alphanumeric Strings

CHAPTER 7

LOOK-UP TABLES 207
Using a More Accurate Sine Look-Up Table – A Paper-Tape Letterer Program

CHAPTER 8

COMMAND DECODERS 235
Single-Letter Command Decoders – A Single-Letter List-Based Command Decoder – A Single-Letter Command Decoder That Uses Two Lists – Multiple-Character Command Decoders – Variable-Length Command Decoders

CHAPTER 9

SYSTEM MONITORS 255
Hardwired Front Panels – General System Monitor Characteristics –A Simple System Monitor Program – A System Monitor for Use With Non-ASCII Keyboards – A System Monitor for Use With a Multiplexed Display – Linking Programs With a System Monitor

CHAPTER 10

BREAKPOINTS AND DEBUGGERS 283
Breakpoints – Breakpoint Instructions – Manually Setting and Clearing the Breakpoint – Automatically Setting and Clearing the Breakpoint – Saving and Printing the Contents of the Registers – Breakpoint Operation – Nondestructively Printing the Contents of the Registers – Adding a Continue Command to the Debugger – Single-Stepping–Executing One Instruction at a Time – Single-Stepping Through Transfer-of-Control Instructions – A Simple Debugger – Final Thoughts on Debuggers

APPENDIX A

THE MK5009 MOS COUNTER TIME-BASE CIRCUIT 335

APPENDIX B

MICROCOMPUTER INTERFACING 339

INDEX 345

List of Program Examples

CHAPTER 1

1-1 Transmitting an ASCII "Z" to an Asychronous Serial Peripheral Device 18
1-2 Transmitting a Character and Waiting for the Transmitter Flag .. 19
1-3 Waiting for the Receiver Flag Before a Character Is Input 21
1-4 A Software-Based Asynchronous Serial Transmission Subroutine .. 27
1-5 Setting the Serial Output Port to a Logic 1 30
1-6 A Software-Based Asynchronous Serial Receiver Subroutine 30
1-7 A Software-Based Asynchronous Serial Transmitter Subroutine for the 8085 Only 38
1-8 A Software-Based Asynchronous Serial Receiver Subroutine for the 8085 Only 40
1-9 A Simple Software-Based Reception and Transmission Test Program 42
1-10 Programming the Mode and Command Registers of the USART . 50
1-11 Accumulator I/O USART Receiver and Transmitter Subroutines . 50
1-12 Initialization Instructions for a Memory-Mapped I/O USART .. 52
1-13 A Memory-Mapped I/O USART Receiver Subroutine 52
1-14 A Memory-Mapped I/O USART Transmitter Subroutine 53

CHAPTER 2

2-1 A Program That Polls Two Keyboards 58
2-2 An ASCII Keyboard Program That Uses a Vectored Interrupt for Service 68
2-3 The Interrupt Service Subroutine That Polls Three Devices 72
2-4 An Improved Three-Device Polled Interrupt Subroutine 74
2-5 Adding a Higher Priority Device to the Interrupt Service Subroutine 76
2-6 Interrupt Service Subroutines for Three Vectored Devices 80
2-7 Programming the Interrupt Mask Register of the 8085 86

CHAPTER 3

3-1 A Real-Time Clock Interrupt Service Subroutine 99
3-2 Programming the Real-Time Clock for 23-ms Operation......... 103
3-3 An Interrupt-Driven Time-of-Day Clock Program 107
3-4 Saving the Time 10:15:00 in R/W Memory 110
3-5 Entering a Time Into the Microcomputer Using a Teletypewriter . 111
3-6 An A.M./P.M. Indicator for the Time-of-Day Clock 116
3-7 An Interrupt Service Subroutine for a 4 × 4 Scanned Keyboard . 120
3-8 A Typical 10-Digit Multiplexed Display Program 124
3-9 An Interrupt-Driven 10-Digit Multiplexed Display 126
3-10 Changing the Current Status Register Before Servicing Interrupt Devices 135

CHAPTER 5

5-1 Finding the Smallest Unsigned Eight-Bit Number in a List 146
5-2 Finding the Largest Unsigned Eight-Bit Number in a List 147
5-3 Finding the Largest and the Smallest Unsigned Eight-Bit Numbers in a List 148
5-4 Finding the Smallest Signed (2s Complement) Eight-Bit Number in a List 150
5-5 Finding the Largest and the Smallest Unsigned 16-Bit Numbers in a List 152
5-6 Finding the Smallest Signed (2s Complement) 16-Bit Number in a List 154
5-7 Finding the Largest and Smallest Unsigned 24-Bit Numbers in a List 156
5-8 Calculating the Node Count From the Initial and Final Addresses of the List 160
5-9 Searching a List of Names and Addresses for the Zip Code, 24060 164
5-10 Printer Instructions for the Zip Code Search Subroutine 167
5-11 Instructions That Print a Carriage Return and a Line Feed 169
5-12 A Comparison of Two Different Up-Arrow Storage Techniques .. 170
5-13 A Program That Is Used To Input and Store Names and Addresses in a List 171
5-14 A Program To Enter the Test Zip Code 174
5-15 A Demonstration of the Zip Code Search Program 177
5-16 Searching for a Zip Code Delimiter 178
5-17 Preventing the Dollar Sign From Being Printed 179

CHAPTER 6

6-1 A List-Sort Subroutine That Uses the Straight Insertion Method . 184
6-2 A List-Sort Subroutine That Uses the Exchange Method (Bubble Sort) 191
6-3 Calculating the Node Count in the ISORT Subroutine (Example 6-1) 192
6-4 Sorting Alphanumeric Strings Using the Exchange Method 198
6-5 A Demonstration Program for the Alphanumeric String-Sort Subroutine (ABSORT) 202
6-6 Using the Demonstration Program With Some Sample Strings ... 204

CHAPTER 7

7-1 Calculating the Sine of an Angle Between 0° and 90° Using a Sine Look-Up Table 211
7-2 Determining the Sine of an Angle Between 0° and 360° Using a Look-Up Table 218
7-3 Modifying the SINANG Subroutine (Example 7-2) To Operate With 16-Bit Sine Values 222
7-4 A Paper-Tape Letterer Program 227
7-5 Simplifying the Paper-Tape Letterer Program 231

CHAPTER 8

8-1 A Single-Letter Command Decoder for a System Monitor 236
8-2 A Flexible Single-Letter Command Decoder 238
8-3 A Two-List Single-Letter Command Decoder 243
8-4 A Four-Letter-per-Command Command Decoder 246
8-5 The Variable-Length Command List Structure 250
8-6 A Variable-Length-Command Command Decoder 252

CHAPTER 9

9-1 A Simple System Monitor That Has Four Commands........... 263
9-2 A Simplified Four-Command System Monitor 269
9-3 A KEYIN Subroutine That Uses a Look-Up Table for Key Code Conversion 272
9-4 A System Monitor That Uses a Multiplexed Display and a Non-ASCII, Nondebounced 12-Key Keyboard 274
9-5 The Output of the System Monitor Command List 280

CHAPTER 10

10-1 Using the OCTIN Subroutine To Input a 16-Bit Address 291
10-2 Using the 8080 To Set the Breakpoint 291
10-3 Setting the Breakpoint and Writing a JMP Instruction at the RST5 Vector Address 293
10-4 A Breakpoint Test Program 294
10-5 Removing a Breakpoint From a Program 295
10-6 Saving the Registers When a Breakpoint Is Reached 295
10-7 Saving and Printing the Registers at TRAP 296
10-8 A Typical Eight-Bit Binary-to-ASCII-Based, Octal Conversion Subroutine 298
10-9 A Program To Test the Breakpoint 299
10-10 The Program With the Breakpoint in Place 300
10-11 Clearing the Breakpoint After It Is Reached 301
10-12 Using Register Pair H To Access the Registers 303
10-13 Printing the Contents of the Registers in the Desired Order When the Breakpoint Is Reached 304
10-14 The Instructions That Cause the 8080 To "Continue" 307
10-15 The New TRAP With Stack-Switching Instructions 308
10-16 CONTIN With Stack-Switching Instructions 309

10-17 A Sample Program To Be Debugged 310
10-18 Determining the Number of Bytes for Each 8080 Instruction 313
10-19 Calculating the Vector Address for a Restart Instruction 318
10-20 Converting All of the Conditional Instructions to Conditional Jump Instructions 320
10-21 A Simple Debugger 323
10-22 Using *DBUG* To Single-Step Through a Program 332

APPENDIX B

B-1 Software Example Showing a Typical Assembler Output 341

1

Asynchronous Serial Communications

If your microcomputer is equipped with only a 16- or 20-key keyboard and some seven-segment displays, it will be time consuming and tedious to enter a 100- or 200-step program into the microcomputer using the keyboard and displays. If a listing of the content of the 100 or 200 memory locations is ever required, it will also be tedious and time consuming to examine memory, one memory location at a time, writing down the content of each memory location as you go. To increase the power and capability of a microcomputer, teletypewriters and cathode-ray terminals (crt's) are often interfaced to the microcomputer system.

There are basically two methods that can be used to interface these communications devices to the microcomputer system. The first method that we will describe requires a large amount of hardware and very little software. The second method that we will describe requires very little hardware but requires many more software instructions. Comparisons of the two methods will be made and the advantages and disadvantages of each method noted.

THE HARDWARE METHOD

A very sophisticated device, the *U*niversal *A*synchronous *R*eceiver/*T*ransmitter (UART), can be used to interface a microcomputer to either a teletypewriter or crt. If you are interested in performing experiments with this device, refer to the book *Interfacing and Scientific Data Communications Experiments.*[1] A more modern device,

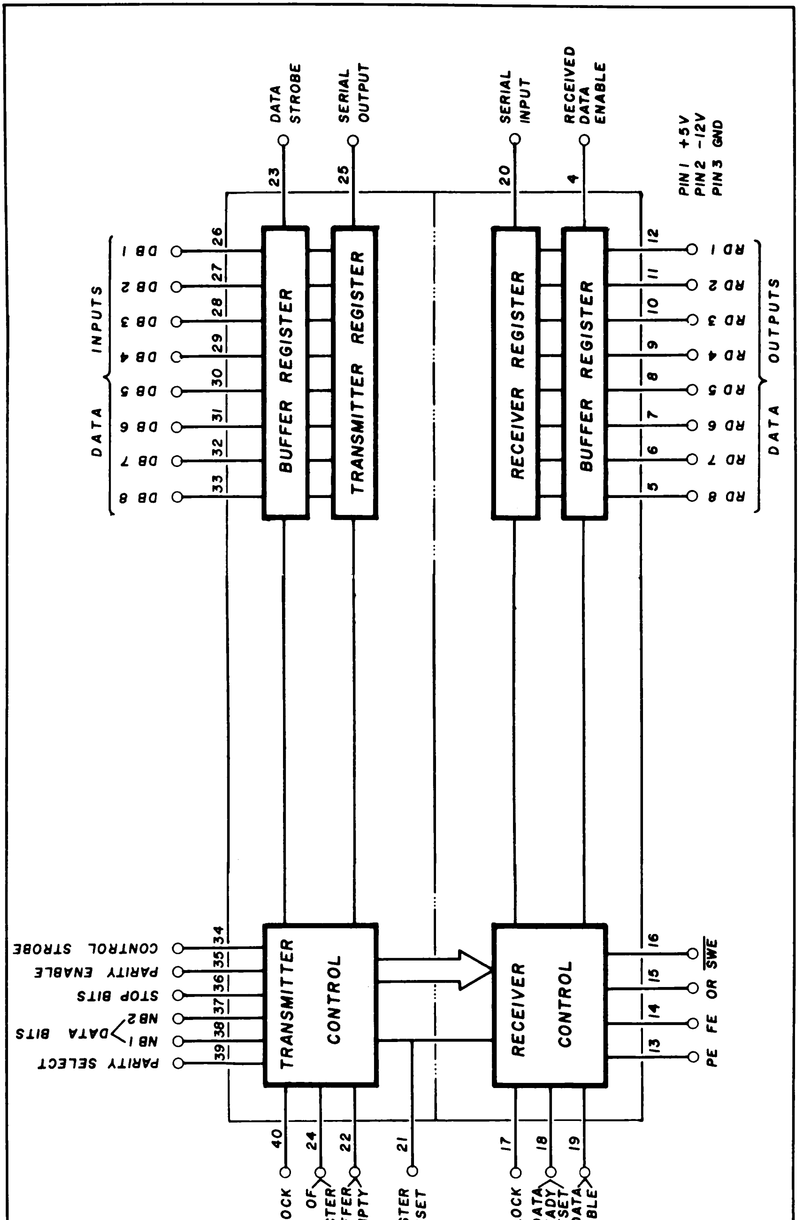

DATA INPUTS
DB 1
DB 2
DB 3
DB 4
DB 5
DB 6
DB 7
DB 8
26
27
28
29
30
31
32
33
BUFFER REGISTER
TRANSMITTER REGISTER
23
DATA STROBE
25
SERIAL OUTPUT
20
SERIAL INPUT
4
RECEIVED DATA ENABLE
PIN 1 +5V
PIN 2 −12V
PIN 3 GND
RECEIVER REGISTER
BUFFER REGISTER
12
11
10
9
8
7
6
5
RD 1
RD 2
RD 3
RD 4
RD 5
RD 6
RD 7
RD 8
DATA OUTPUTS
CONTROL STROBE
PARITY ENABLE
STOP BITS
NB 2
NB 1
DATA BITS
PARITY SELECT
34
35
36
37
38
39
TRANSMITTER CONTROL
RECEIVER CONTROL
16
15
14
13
SWE
OR
FE
PE
40
24
22
21
17
18
19
CLOCK
END OF CHARACTER
BUFFER EMPTY
MASTER RESET
CLOCK
DATA READY RESET
DATA AVAILABLE

Fig. 1-1. The pin assignments for a 40-pin UART.

Table 1-1. The Capabilities of a UART

- Independent receiver and transmitter.
- Receive and transmit five-, six-, seven-, or eight-bit data words.
- Even, odd, or no parity can be selected.
- Operates from dc to at least 20,000 bits per second.
- Three receiver error flags.
- Three-state outputs.

the *U*niversal *S*ynchronous/*A*synchronous *R*eceiver/*T*ransmitter (USART), is also finding a large number of applications in the computer-based communications field.

A UART is packaged in a 40-pin integrated circuit. The pin assignments for one of the popular UART devices is shown in Fig. 1-1. The UART contains a completely independent asynchronous serial receiver and asynchronous serial transmitter. This means that it can receive data from a keyboard of a crt at one speed and *simultaneously* transmit *different* data to a teletypewriter printer at a *different* speed. Some of the other features of the UART are listed in Table 1-1. The corresponding CMOS UARTs operate from single power supplies and dissipate less power.

UART FEATURES

As can be seen in Table 1-1, the UART can transmit and receive five, six, seven, or eight bits of information. For the UART to operate with these word lengths, it must be *programmed* for the desired word length. This programming is performed by placing a logic 1 or a logic 0 on UART pins 37 and 38, as summarized in Table 1-2. This programming will determine the word length for *both* the receiver and transmitter sections of the UART.

Two of the UART pins are also dedicated to parity generation and selection. If parity is to be used, either even or odd parity can be selected. To eliminate the parity bit from the transmitted or received data word, pin 35 of the UART must be at a logic 1. If this pin is held at a logic 0, then the parity, as determined by the logic state of pin 39, will be used in *both* the transmitter and receiver sections of the UART (Table 1-3).

Table 1-2. Programming the Word Length of the UART

Pin 37	Pin 38	Word Length (Bits per Word)
0	0	5
0	1	6
1	0	7
1	1	8

Table 1-3. Programming the Parity of the UART

Pin 35	Pin 39	Parity Selected
0	0	Odd Parity
0	1	Even Parity
1	0	No Parity
1	1	No Parity

The number of stop bits can also be programmed into both the receiver and transmitter sections of the UART. This is controlled by the logic level applied to pin 36. If pin 36 is connected to a logic 0, then one stop bit will be transmitted; the receiver will also expect to receive only one stop bit. If a logic 1 is applied to pin 36, the transmitter will transmit two stop bits, and the receiver will expect two stop bits in the received word.

One important point should be made about programming the UART with the word length, parity, and the number of stop bits. *These logic levels will not be loaded into an internal UART holding register until pin 34 is taken to a logic 1.* This pin can be permanently wired to a logic 1 level, or it can be pulsed with an output pulse so that the content of the data bus is loaded into this holding register. The pulse must have a minimum duration of 300 nanoseconds (ns).

Finally, there are three error flags associated with the receiver section of the UART. These flags indicate a *parity error* (PE, pin 13), a *framing error* (FE, pin 14), and an *overrun error* (OE, pin 15). A parity error occurs when the parity of the received data word does not agree with the parity programming of the UART. This flag goes to a logic 1 if a parity error occurs. A framing error occurs, as indicated by the FE flag going to a logic 1, if the received character does not have a valid stop bit or the proper number of stop bits. The *overrun flag* (OR) goes to a logic 1 if the *received data available flag* (pin 19) has not been reset after the receipt of the latest data word. In other words, the overrun flag goes to a logic 1 if one data word runs over another data word in the receiver. These flags are buffered by three-state buffers contained on the UART integrated circuit, so these pins can be directly wired to the microcomputer data bus. The flags are gated onto the data bus when pin 16, the *status word enable* ($\overline{\text{SWE}}$) pin, is taken to a logic 0. This pin could be pulsed by the proper combination of $\overline{\text{IN}}$ ($\overline{\text{I/O R}}$) or $\overline{\text{MEMR}}$ with a device address.

SERIAL DATA FORMAT

In general, most teletypewriters and crt's are connected to the UART or USART in the interface with only four wires. How can

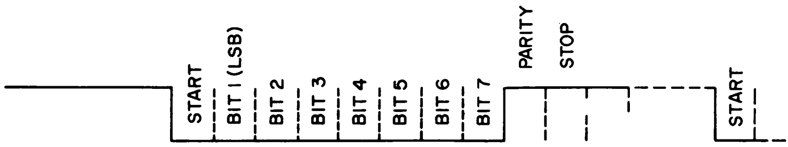


Fig. 1-2. A serial data stream for a seven-bit character plus a parity bit.

five-, six-, seven-, or eight-bit characters be transmitted or received over just four wires? In practice, two of the wires are used by the UART to transmit data to the peripheral device and two wires are used to receive data from the peripheral device. To transmit data over two wires, the data words are transmitted or received a *single bit at a time.* This is a *serial* method of communication. A seven-bit character serial data stream is shown in Fig. 1-2.

This serial method of communicating is used by the transmitters in both the UART and the peripheral device (i.e., a teletypewriter or crt). The UART and the teletypewriter or crt must also be capable of receiving serial data.

The UART, because it is such a sophisticated device, can actually transmit and receive serial characters. These characters are further characterized as being *asynchronous serial,* which means that there is no relationship between when a character is transmitted or received and a clock signal. If characters are received or transmitted in a synchronous serial fashion, then the characters are transmitted and received with a specific relationship to a clock signal.

HARDWARE UART SOFTWARE

As you can see by examining many of the examples in the book *8080/8085 Software Design—Book 1*[2], it is very easy to transmit a character to a teletypewriter, or receive a character from a teletypewriter, when the teletypewriter is interfaced to the microcomputer with a UART. In Chapter 3 of *8080/8085 Software Design—Book 1*[2], we assumed that a teletypewriter was interfaced to the 8080 microcomputer using a UART. To transmit a character to an asynchronous serial peripheral device, the 8080 simply has to output the content of the A register to the UART (assuming that it is an accumulator I/O device). The bidirectional data bus of the microcomputer is connected directly to the transmitter input pins of the UART*. When the proper OUT instruction is executed, pin 23 of the UART must be strobed with a negative pulse so that the data word on the data bus is strobed into the transmitter section of the UART. When this occurs, the UART automatically begins the serial transmission of the character to the peripheral device. Therefore, to transmit the ASCII character "Z" to the peripheral device, the program listed in Example 1-1 can be executed.

---

* This assumes that the three-state outputs of the UART are compatible with the *timing* requirements of the microcomputer. This is not always the case. Check specific device specification sheets for the three-state output timing requirements.

Example 1-1: Transmitting an ASCII "Z" to an Asynchronous Serial Peripheral Device

```
•
•
MVIA    /LOAD THE A REGISTER WITH THE ASCII
132     /CHARACTER "Z" (132 = HEX 5A).
OUT     /OUTPUT THE CHARACTER TO THE UART (STROBE
001     /THE DATA STROBE PIN).
•
•
```

Of course, after the OUT instruction is executed, the 8080 could load the UART with another character to be transmitted to the peripheral device. However, as we have seen previously, teletypewriters and crt's require a fixed amount of time to actually receive the character and print it. For a teletypewriter, this time is about 100 milliseconds (ms). To prevent the 8080 from transmitting characters faster than the teletypewriter can receive them, there is a *flag* that is generated by the UART that indicates when the 8080 can output another character to the UART. When this flag is a logic 1, the complete character has been transferred to the transmitter register of the UART and another character can then be loaded into the buffer register of the transmitter. After the first character has been completely transmitted, the next character is transferred from the buffer register to the transmitter register, where it is then transmitted serially.

This means that the 8080 must monitor the state of the *buffer empty* flag of the UART transmitter to determine when another character can be output to the UART. This flag is brought out to pin 22 of the UART integrated circuit. This pin is normally in the three-state condition (the high-impedance state), but when the *status word enable* pin (pin 16) is brought to a logic 0, the state of the flag can be monitored at pin 22. This means that pin 22 can be directly connected to the bidirectional data bus of the 8080 microcomputer. By executing an appropriate IN instruction and by gating the $\overline{\text{IN}}$ signal ($\overline{\text{I/O R}}$) and an appropriate device address, the state of this flag can be input into the A register of the 8080. A subroutine that actually outputs a character to a UART and, before returning, waits for this flag to be a logic 1, is shown in

Example 1-2. This type of subroutine has been used many times in the book *8080/8085 Software Design—Book 1*[2].

Example 1-2: Transmitting a Character and Waiting for the Transmitter Flag

```
/TRANSMIT AN ASCII "Z" TO THE TELETYPEWRITER
/OR CRT AND THEN WAIT FOR THE TRANSMITTER FLAG.

ATRANS, MVIA    /LOAD A WITH THE ASCII CHARACTER
        132     /"Z" (132 = HEX 5A).
        OUT     /OUTPUT IT TO THE UART. THE UART
        100     /AUTOMATICALLY BEGINS THE TRANSMISSION.
WAIT,   IN      /INPUT THE DATA WORD THAT CONTAINS THE
        101     /TRANSMITTER FLAG OF THE UART.
        ANI     /SAVE ONLY THE TRANSMITTER FLAG
        040     /IN A (040 = HEX 20).
        JZ      /THE FLAG IS STILL ZERO, SO
        WAIT    /CONTINUE WAITING FOR IT TO
        0       /GO TO A LOGIC ONE.
        RET     /THEN RETURN FROM THIS SUBROUTINE.
```

One of the best features of the UART is the fact that it can be used to transmit data to a 10-character-per-second teletypewriter and at the same time receive data from a 240-character-per-second crt. This can be done since the rate at which bits of information are received and transmitted by the UART is determined by two separate TTL-compatible clocks. The clock for the transmitter section of the UART must be applied to pin 40 and the clock for the receiver must be applied to pin 17.

One of the terms often used to describe asynchronous communications is the *bit rate* (*baud*) at which information can be transmitted or received. The bit rate, when divided by the number of bits per character, determines the number of characters per second that can be received or transmitted. As we already know, the UART can send or receive five-, six-, seven-, or eight-bit characters. However, a UART must also transmit a *start bit* and at least one *stop bit.* Therefore, for an eight-bit character, the number of bits shown in Table 1-4 may be transmitted to the peripheral device. As you can see from this table, the longest character that can be transmitted or received contains 12 bits of information. *No matter what the length of the data word being transmitted* (*received*), *a start bit*

Table 1-4. The Number of Bits That May Be Transmitted By a UART

One start bit + eight-bit data word + one stop bit.
One start bit + eight-bit data word + two stop bits.
One start bit + eight-bit data word + one parity bit + one stop bit.
One start bit + eight-bit data word + one parity bit + two stop bits.

Table 1-5. Common Bit Rates and the Required Clock Frequencies

Bit Rate (bits/s)	Clock Frequency (kHz)	Clock Period (μs)
110	1.76	586
150	2.40	417
300	4.80	208
600	9.60	104
1,200	19.2	52.1
2,400	38.4	26.0
4,800	76.8	13.0
9,600	153.6	6.51
19,200	307.2	3.26

and at least one stop bit must also be transmitted (received). As you can see, you must be careful when comparing the rate at which characters are received or transmitted between one microcomputer system and another. Because of the internal organization of the UART, the transmitter and receiver clock frequencies must be 16 times the desired bit rate. Therefore, the common bit rates and required clock frequencies are listed in Table 1-5.

The operation of the receiver in the UART is just the reverse of the operation of the transmitter. The receiver must be capable of receiving a character, one bit at a time. When the required number of data bits is received, a five-, six-, seven-, or eight-bit parallel data word must be formed so that the 8080 can input the received characters into one of the general-purpose registers. If the UART is interfaced to the microcomputer using accumulator I/O techniques, the character would have to be input into the A register. If memory-mapped I/O interfacing techniques are used, the 8080 can input the data word into any of the general-purpose registers. Referring to the pinout diagram of the UART (Fig. 1-1), the character that the UART receives will be placed on pins 5 through 12 when pin 4, the *received data enable* (RDE) pin, is brought to a logic 0. If a logic 1 is applied to pin 4, pins 5 through 12 will be in the three-state condition (high-impedance state).

The receiver pins of the UART can be wired directly to the bidirectional data bus of the 8080 microcomputer. The transmitter pins can also be directly wired to the bidirectional data bus. To actually bring pin 4 to a logic 0 so that the received character is read by the 8080, an IN instruction could be executed. This means that $\overline{\text{IN}}$ ($\overline{\text{I/O R}}$) is gated with a decoded device address (active low) with an OR gate. The output of the OR gate will go to the logic 0 state when an IN instruction is executed that contains the selected device address of the UART. At that time, the data received by the UART will be gated onto the bidirectional data bus and into the A register of the 8080.

Example 1-3: Waiting for the Receiver Flag Before a Character Is Input

```
/MONITOR THE RECEIVER FLAG OF THE UART.
/WHEN IT IS A LOGIC ONE, A CHARACTER
/HAS BEEN RECEIVED AND IT CAN BE INPUT.

RCVR,   IN      /INPUT THE DATA WORD THAT CONTAINS
        101     /THE RECEIVER FLAG OF THE UART.
        ANI     /SAVE ONLY THE RECEIVER FLAG
        020     /IN A (020 = HEX 10).
        JZ      /THE FLAG IS STILL ZERO, SO
        RCVR    /CONTINUE WAITING FOR IT TO
        0       /GO TO A LOGIC ONE.
        IN      /THE FLAG IS A LOGIC ONE, SO
        100     /INPUT THE CHARACTER FROM THE UART
        RET     /AND RETURN WITH IT IN A.
```

When a character is transmitted by the UART, the 8080 must be programmed to wait for the transmitter *buffer empty* flag to go to a logic 1 before another character can be transmitted. The 8080 must also be programmed to monitor the *received data available* flag (pin 19) which is associated with the receiver section of the UART. When this flag is a logic 1, the UART has received a character and the character can be input by the 8080. If the flag is a logic 0, the UART has not yet received a character. The software that actually waits for this flag before the character is input is simple, as shown in Example 1-3.

Not only must the software sense that the flag is a logic 1, but there also has to be some way of clearing the flag back to the logic 0 state. If the flag cannot be cleared, the 8080 will sense and input the same character a number of times, even though only one character was transmitted to the UART by the peripheral device. The flag can be cleared to the logic 0 state by applying a negative pulse to pin 18 (*data ready reset*), which resets the *data available* flag.

By means of a clever hardware design, the second IN instruction in Example 1-3 can be used to input not only the character that is received by the UART, but it can also be used to pulse pin 18 of the UART, which clears the *data available* flag. This means that the 8080 will input each character received only once, each time a key on the keyboard on the teletypewriter or crt is pressed. The schematic of a UART-based asynchronous communications interface is shown in Fig. 1-3. The device address decoder portion of this interface is shown in Fig. 1-4.

Note that this interface uses a split-bus design, similar to the S-100–type bus. The three-state outputs of the UART have been buffered with an additional set of three-state buffers. This was necessary, since the UART outputs were too "slow" to be compatible with the 8080 timing.

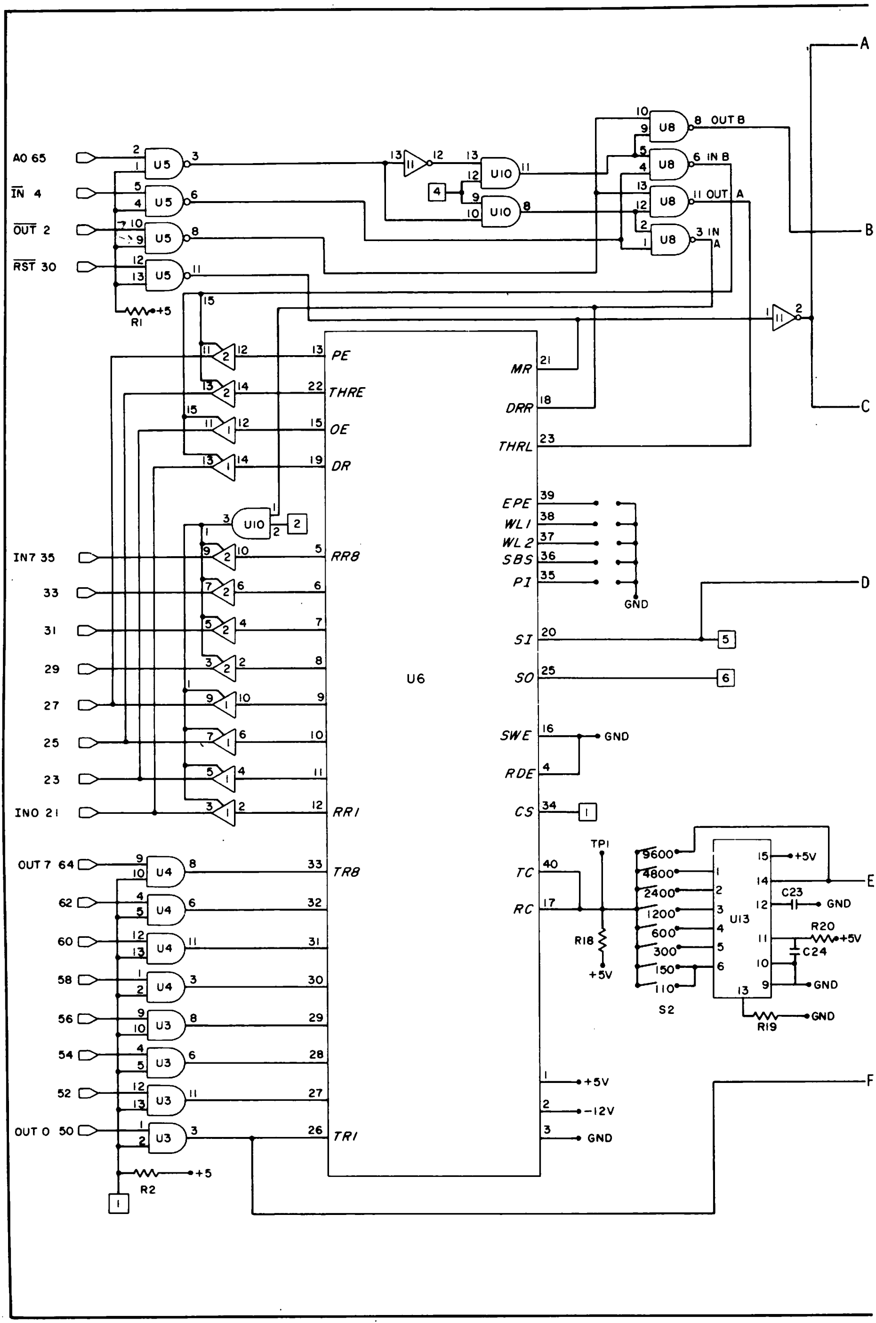
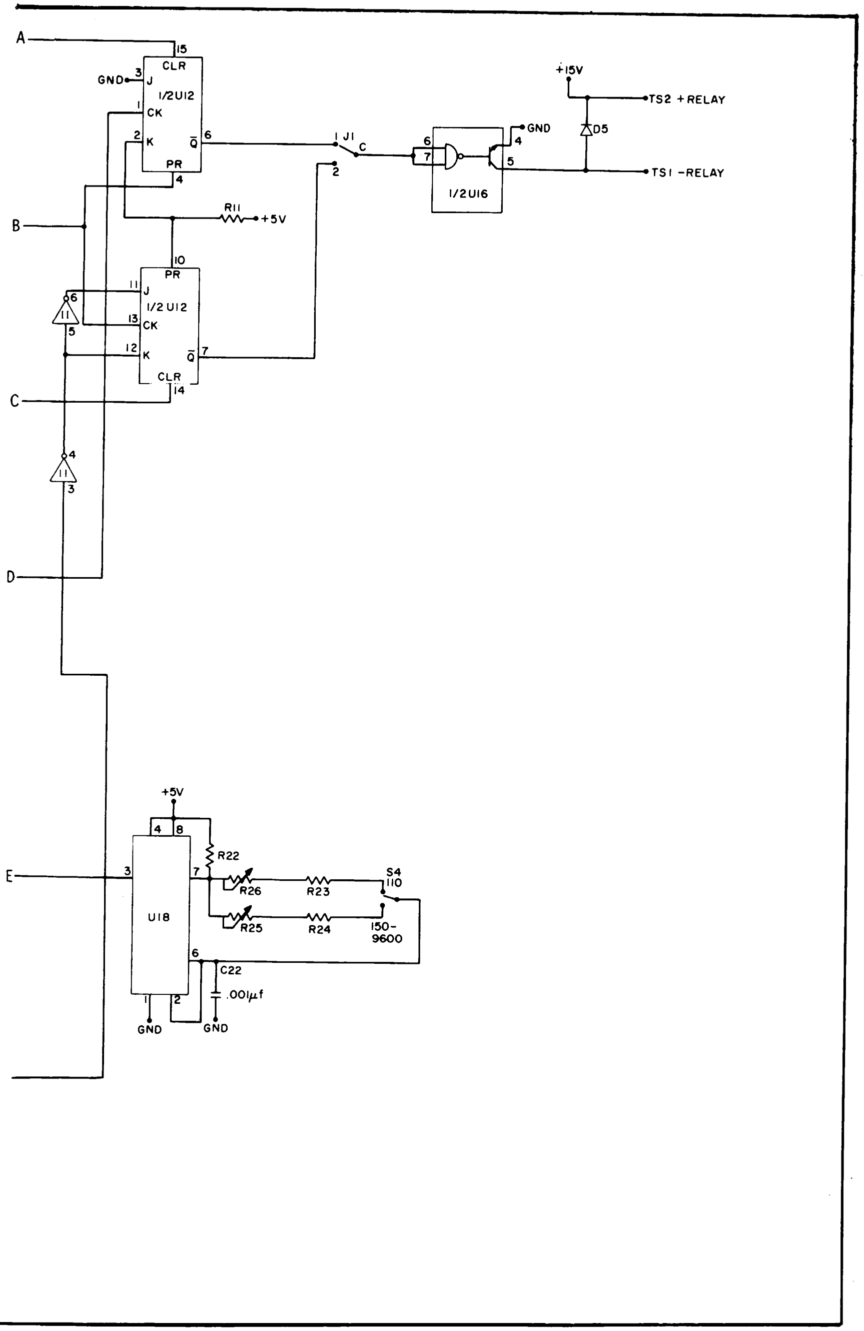

A0 65
IN 4
OUT 2
RST 30
U5
R1
+5
U11
U10
4
U8
OUT B
IN B
OUT A
IN A
A
B
C
D
E
F
U6
PE
THRE
OE
DR
RR8
RR1
TR8
TR1
MR
DRR
THRL
EPE
WL1
WL2
SBS
PI
GND
SI
SO
SWE
RDE
CS
TC
RC
5
6
1
2
IN7 35
33
31
29
27
25
23
IN0 21
OUT 7 64
62
60
58
56
54
52
OUT 0 50
U4
U3
R2
+5V
−12V
TP1
R18
9600
4800
2400
1200
600
300
150
110
S2
U13
C23
R20
C24
R19

Fig. 1-3. A UART-based asynchronous


A
B
C
D
E
15
CLR
3
GND
J
1/2U12
1
CK
2
K
6
Q̄
PR
4
R11
+5V
10
PR
11
J
1/2 U12
13
CK
12
K
Q̄
7
CLR
14
6
11
5
4
11
3
1 J1
C
2
6
7
1/2U16
GND
4
5
+15V
TS2 +RELAY
D5
TS1 -RELAY
+5V
4
8
R22
3
7
U18
R26
R23
S4
110
R25
R24
150-
9600
6
C22
.001μf
1
2
GND
GND

communications interface.

What are the advantages and disadvantages of using UARTs? Even though the UART is a powerful integrated circuit, it still requires a number of additional integrated circuits to complete the interface between the teletypewriter or crt and the 8080-based microcomputer. In Fig. 1-3, the UART used is a General Instruments Corp. AY-5-1013. This UART requires an extra power supply (−12 volts), which may be a costly addition to one of the microcomputer systems that requires only +5 volts. However, UARTs are available (at slightly higher cost) that require only +5 volts (i.e., The Western Digital TR1863). Additional circuitry is also required (shown in Fig. 1-4) to convert the TTL input and output logic levels of the UART to the 20 mA or EIA Standard RS-232C required by many peripherals.

If you are interested in using UARTs for communicating with asynchronous serial peripheral devices such as teletypewriters and crt's, refer to *Interfacing and Scientific Data Communications Experiments*[1]. Not only does this book explore the many features of UARTs in greater depth, but there are also experiments that you can actually perform so that you can become proficient at using UARTs.

SOFTWARE-BASED UART'S

To a teletypewriter or crt, a "software-based" UART must look exactly like a UART chip. As long as the advantages and disadvantages of both methods are realized, no problems will occur in the use of either method. The software-based UART differs from the UART chip in that a number of software instructions are used to replace the 40-pin integrated circuit.

These instructions simply cause serial 1s and 0s to be transmitted to, or received from, a teletypewriter or crt or other asynchronous serial device. When a UART integrated circuit is used, it can be electronically programmed to transmit and/or receive five-, six-, seven-, or eight-bit characters. An even or parity bit can also be added to the transmitted character and detected in a received character. By applying the proper voltages to these programming pins, the parity bit can also be eliminated. With a software-based UART, instructions must be added or changed in the transmitter and receiver subroutines so that five-, six-, seven-, or eight-bit characters can be transmitted and/or received. Also, the 8080 has to be programmed to transmit (receive) characters with even parity, odd parity, or no parity. For simplicity, the UART software will be written to transmit and receive a start bit, an eight-bit data word, and two stop bits. No parity will be used. This means that the 8080 will be programmed to transmit and receive 11-bit words.

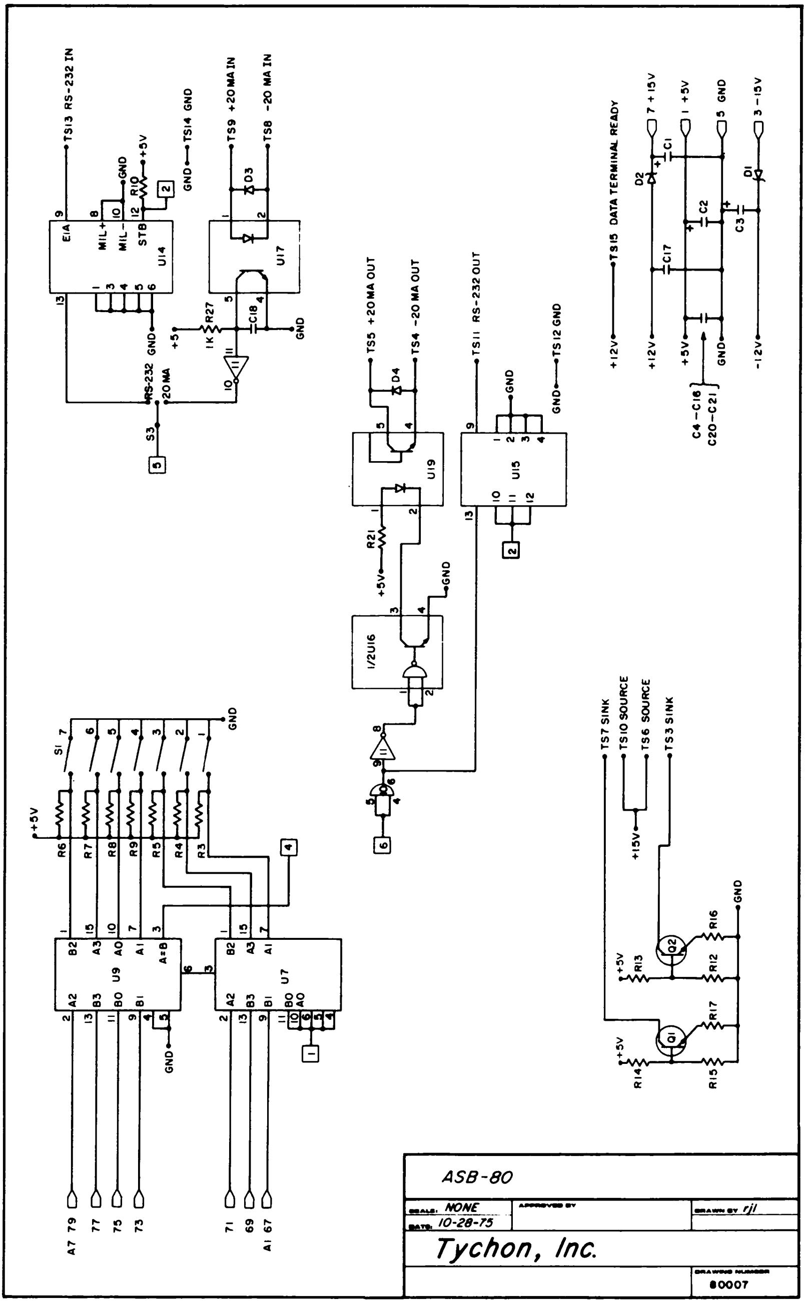

ASB-80
SCALE: NONE
DATE: 10-28-75
APPROVED BY
DRAWN BY rjl
Tychon, Inc.
DRAWING NUMBER
80007
TS13 RS-232 IN
TS14 GND
TS9 +20 MA IN
TS8 -20 MA IN
TS5 +20 MA OUT
TS4 -20 MA OUT
TS11 RS-232 OUT
TS12 GND
TS15 DATA TERMINAL READY
TS7 SINK
TS10 SOURCE
TS6 SOURCE
TS3 SINK
U14
U17
U19
U15
1/2U16
U9
U7
S1
S3
RS-232
20 MA
+5V
+12V
-12V
GND
7 +15V
1 +5V
5 GND
3 -15V
C4-C16
C20-C21
A7 79
77
75
73
71
69
A1 67

Fig. 1-4. The device address decoder for the asynchronous communications interface.

The interface electronics that are used with the software-based UART subroutines are very simple (Fig. 1-5). The serial input and output lines in this interface have the same functions as the serial input and output pins on the UART integrated circuit. Some additional circuitry would be required to convert the TTL input and output logic levels of this interface to the 20 mA or RS-232C required by the crt or teletypewriter. However, we estimate that about eight integrated circuits can be eliminated, including the 40-pin UART integrated circuit, by using a software serialization technique. By eliminating these integrated circuits, a more complex sequence of assembly language instructions is needed so that the 8080 can communicate with asynchronous serial devices.

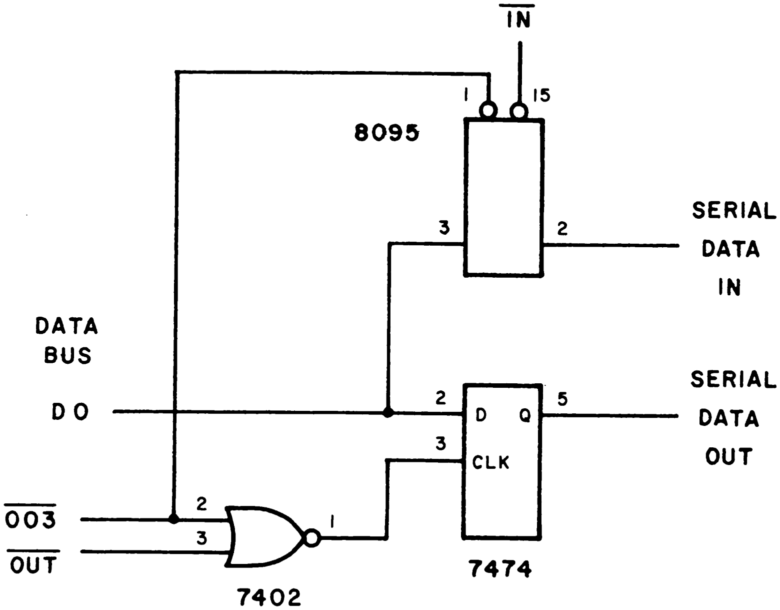


Fig. 1-5. A software-based asynchronous communications interface.

A simple software serialization and transmission subroutine is listed in Example 1-4. This subroutine performs many of the same functions as the transmitter within the UART. The subroutine listed in Example 1-4 must only be called with the eight-bit character, which is to be transmitted, present in the A register.

This subroutine first saves register pair H and then loads the L register with 013 (decimal 11, hexadecimal 0B), which is the number of start bits (one), data bits (eight), and stop bits (two) that will be transmitted. The contents of the A register are then ORed to-

Example 1-4: A Software-Based Asynchronous Serial Transmission Subroutine

```
/THIS SUBROUTINE USES A SINGLE-BIT OUTPUT PORT
/TO TRANSMIT ASYNCHRONOUS CHARACTERS. ENTER
/THE SUBROUTINE WITH THE EIGHT-BIT CHARACTER TO
/BE TRANSMITTED IN A.

TRANS,   PUSHH    /SAVE REGISTER PAIR H ON THE STACK.
         MVIL     /LOAD THE L REGISTER WITH THE
         013      /NUMBER OF BITS TO BE TRANSMITTED.
         ORAA     /KEEP A THE SAME, BUT CLEAR THE CARRY TO 0.
         RAL      /ROTATE THE CARRY INTO D0 OF A.
NXTBIT,  OUT      /OUTPUT D0.
         003
         CALL     /CALL THE DELAY SUBROUTINE THAT
         DELAY    /PRODUCES A DELAY OF ONE BIT TIME.
         0
         RAR      /NOW ROTATE D1 INTO D0.
         STC      /SET THE CARRY FOR THE STOP BITS.
         DCRL     /DECREMENT THE BIT COUNT.
         JNZ      /IF THE COUNT IS NONZERO,
         NXTBIT   /JUMP BACK AND TRANSMIT ANOTHER
         0        /DATA BIT.
         POPH     /RESTORE H AND L TO THEIR ORIGINAL VALUES,
         RET      /THEN RETURN FROM THE SUBROUTINE.

DELAY,   PUSHPSW /SAVE THE FLAGS AND A ON THE STACK.
         PUSHH    /SAVE H AND L ON THE STACK.
         LXIH     /LOAD REGISTER PAIR H WITH
         365      /A 16-BIT TIMING BYTE.
         002
WAIT,    DCXH     /DECREMENT THE TIMING BYTE BY ONE.
         MOVAH    /MOVE THE MSBY INTO A.
         ORAL     /OR IT WITH THE LSBY.
         JNZ      /IF THE 16-BIT NUMBER IS NON-
         WAIT     /ZERO, THEN JNZ TO THE DCXH
         0        /INSTRUCTION.
         POPH     /POP H AND L AND THE PSW OFF OF THE STACK.
         POPPSW
         RET      /RETURN FROM THE DELAY SUBROUTINE.
```

gether. This does not change the contents of the A register, but it does clear the carry to 0. This logic 0 will be used as the start bit, which will be transmitted for one *bit time,* or the time required to transmit one bit of information. Remember, a start bit *must* be transmitted at the beginning of each data word that is transmitted serially.

After the ORAA instruction is executed, the contents of the A register and the carry are rotated left once. This rotates the carry (a logic 0) into bit D_0 of the A register. The OUT instruction then places the eight-bit contents of the A register on the data bus, but only bit D_0 is latched by the SN7474 D flip-flop in the interface

(Fig. 1-5). After the start bit is latched by the D flip-flop, the DELAY subroutine is called.

The DELAY subroutine generates a delay of one bit time. How long is this delay? The duration of the time delay generated by this subroutine is determined by the desired rate at which the 8080 must transmit information to the peripheral device. If the 8080 is communicating with a 10-character-per-second teletypewriter, a transmission rate of 110 bits per second (remember, 11 bits per character) is required. Therefore, the bit time is 9.09 ms. This means that the start bit, eight data bits, and two stop bits must all be latched by the D flip-flop for 9.09 ms (9090 μs). In Example 1-4, the DELAY subroutine generates a delay of 9.09 ms. We have assumed that the 8080 is transmitting characters to a 110-bit-per-second teletypewriter and that the 8080 has a cycle time of 500 ns.

When the DELAY subroutine is executed, both register A and the flags, as well as register pair H, are saved on the stack. A 16-bit timing byte is then loaded into register pair H. The 8080 decrements the content of register pair H until it is zero. When it is zero, the 8080 restores the registers and flags and returns to the transmitter subroutine. Why is the PSW saved on the stack at the beginning of this subroutine? This is done because the A register and the carry contain the information that is being transmitted to the peripheral device. Therefore, this information must be saved on the stack.

When the 8080 returns from the DELAY subroutine, it has been transmitting the start bit for 9.09 ms. The 8080 then rotates the contents of the A register and the carry one bit to the right. This rotates bit D_0 back into the carry and bit D_1 into bit D_0, etc. The A register now contains the original eight-bit character that must be transmitted to the peripheral device. The carry is then set to a logic 1 when the STC instruction is executed. The bit count (the number of bits of information being transmitted) contained in the L register is then decremented. If the result of the DCRL instruction is nonzero, the JNZ to NXTBIT is executed. At NXTBIT, the content of bit D_0 is output and latched by the D flip-flop in the interface. The DELAY subroutine is then called, so that bit D_0 is transmitted to the peripheral for 9.09 ms.

Since the content of the A register is being transmitted a bit at a time, is the LSB or MSB of register A transmitted first? After the start bit (which is always 0) has been transmitted, the LSB in register A is transmitted first. The MSB in register A is the last data bit to be transmitted. If required, the software can be altered so that the MSB is transmitted first. However, almost all teletypewriters and crt's require that the LSB of the data word be transmitted to them first.

Table 1-6. The Contents of the A Register and the Carry During the Transmission of a Character

Carry	A	Bit
0	10001110	Start Bit
1	01000111	Data—LSB
1	10100011	Data
1	11010001	Data
1	11101000	Data
1	11110100	Data
1	11111010	Data
1	11111101	Data
1	11111110	Data—MSB
1	11111111	Stop Bit
1	11111111	Stop Bit

Let us examine how the two stop bits are generated. After transmitting the full eight-bit data word, register A contains 377 (FF), since the carry is set to a logic 1 after each rotation and is rotated into the A register by the RAR instruction. Suppose the character 107 (47) is to be transmitted to the peripheral device. The contents of the A register and the carry, each time the OUT instruction is executed, are shown in Table 1-6. When the eight-bit contents of the A register have been transmitted, the L register (the bit count) still contains 002 (02). Therefore, two additional bits must be transmitted. The subroutine always transmits bit D_0 of the A register, so the 1s that were rotated from the carry into the A register actually form the two stop bits. Therefore, the last two times through the transmission loop, register A contains 377 (FF), and two logic 1s are transmitted to the peripheral device as stop bits.

One final point about the software-based asynchronous serial "transmitter." When no characters are being transmitted by the UART (hardware or software-based) to a peripheral device, the output of the UART should be a logic 1. For a 20-mA current loop, this logic 1 causes current to flow through the loop. If the peripheral device uses the RS-232C convention, then the logic 1 output of the UART must cause a voltage level of between +5 and +15 volts to be output to the peripheral device. When power is applied to a UART chip and it is reset, the serial output of the UART becomes a logic 1. When power is applied to the circuit shown in Fig. 1-5, the SN7474 D flip-flop will output either a logic 1 *or* a logic 0. There is no way of knowing which level will be output. Therefore, at the start of *any* program that uses such a software-based serialization scheme, the instructions listed in Example 1-5 should be executed. These instructions load the D flip-flop with a logic 1, so that it can be transmitted to the peripheral device after the proper conversion (TTL to 20 mA or TTL to RS-232C).

Example 1-5: Setting the Serial Output Port to a Logic 1

```
START,   LXISP   /LOAD THE STACK POINTER WITH A R/W
         STACK   /MEMORY ADDRESS.
         0
         MVIA    /THEN LOAD BIT D0 OF THE A
         001     /REGISTER WITH A LOGIC ONE.
         OUT     /OUTPUT BIT D0 TO THE D
         003     /FLIP-FLOP.
          •      /THEN EXECUTE THE REMAINDER OF THE
          •      /PROGRAM.
```

As you have seen, the transmission of an eight-bit asynchronous serial character is relatively easy using this technique. The subroutine that enables the 8080 to receive asynchronous serial characters is only slightly more complex.

The hardware required for the software-based receiver function consists of the one-bit input port shown in the previous schematic (Fig. 1-5). When the appropriate input instruction is executed, one bit of information will be input into bit D_0 of the A register. Just as events had to be accurately timed during the transmission of asynchronous serial data, they also have to be accurately timed so that the 8080 can properly receive asynchronous serial data.

When the instructions in Example 1-6 are executed, the 8080 expects to receive a serial "stream" of bits containing a start bit, eight data bits (LSB first), and then two stop bits. Thus, the 8080 first tests the one-bit input port for a logic 0 condition. This is done to detect the start of the start bit being transmitted to the 8080 microcomputer. Therefore, the 8080 inputs the information into bit D_0 of the A register. The ANI instruction sets all of the other bits in the A register to 0. If the content of the A register is nonzero, then no start bit is being received, and the JNZ to RCVR is executed. When a logic 0 is detected, the HALF subroutine, which generates a time delay of one-half of a bit time, is called. If data is to be received at the rate of 110 bits per second, the bit time is 9.09 ms and the half-bit time is 4.54 ms.

Example 1-6: A Software-Based Asynchronous Serial Receiver Subroutine

```
/THIS SUBROUTINE RECEIVES EIGHT-BIT CHARACTERS
/USING A SINGLE-BIT INPUT PORT. THE 8080
/WILL EXIT THIS SUBROUTINE WITH THE RECEIVED
/CHARACTER IN THE B REGISTER. THE PARITY OF
/THE CHARACTER IS NOT CHECKED.

RCVR,    IN      /INPUT THE DATA FROM THE SINGLE-BIT
         003     /INPUT PORT (003 = HEX 03).
         ANI     /SAVE ONLY THE DATA BIT IN D0
         001     /(001 = HEX 01).
         JNZ     /NOTHING IS BEING TRANSMITTED, NOT
         RCVR     /EVEN A START BIT, SO KEEP WAITING
         0        /FOR THE START BIT.
         CALL     /A ZERO WAS DETECTED, SO WAIT FOR ONE-
         HALF     /HALF A BIT TIME.
         0
         IN       /THEN TEST THE DATA LINE AGAIN TO
         003      /ENSURE THAT A START BIT WAS REALLY
         ANI      /DETECTED.
         001
         JNZ      /IT WAS NOISE, SO JUMP BACK TO
         RCVR     /THE BEGINNING OF THE SUBROUTINE.
         0
         LXIB     /IT WAS A VALID START BIT, SET B TO
         010      /000 FOR TEMPORARY STORAGE AND C TO
         000      /010 BECAUSE IT IS THE BIT COUNTER.
NXTBIT,  CALL     /WAIT FOR ONE BIT TIME BEFORE
         DELAY    /A DATA VALUE IS INPUT
         0        /INTO BIT D0 OF THE A REGISTER.
         IN       /RIGHT IN THE MIDDLE OF THE BIT, SO
         003      /INPUT IT
         ANI      /AND SAVE ONLY THE SINGLE DATA BIT.
         001
         ADDB     /ADD THE TEMPORARY WORD TO THE BIT.
         RRC      /ROTATE IT TO THE RIGHT ONE POSITION.
         MOVBA    /THEN SAVE THE RESULT IN B.
         DCRC     /DECREMENT THE BIT COUNTER.
         JNZ      /IT IS NONZERO, SO GET ANOTHER BIT.
         NXTBIT   /BUT FIRST WAIT AN ENTIRE BIT TIME
         0        /BEFORE INPUTTING THE DATA BIT.
         MOVAB    /GET THE CHARACTER.
         RLC      /ROTATE IT ONCE TO THE LEFT
         MOVBA    /AND PUT IT BACK IN B.
         CALL     /ALL THE DATA BITS HAVE BEEN READ, SO
         DELAY    /WAIT FOR THE LAST DATA BIT TO GO BY
         0        /AND THEN THE STOP BITS.
         CALL
         DELAY    /(A DELAY OF 2 BIT TIMES HAS
         0        /NOW BEEN GENERATED)
         RET      /RETURN WITH THE CHARACTER IN B.

DELAY,   CALL     /TO GENERATE A DELAY OF ONE
         HALF     /BIT TIME, EXECUTE THE HALF SUB-
         0        /ROUTINE TWICE.
HALF,    PUSHH    /SAVE H AND L ON THE STACK.
         LXIH     /LOAD REGISTER PAIR H WITH A 16-BIT
         172      /TIMING BYTE WHICH IS ONE-HALF
         001      /THE BIT RATE.
WAIT,    DCXH     /DECREMENT THE TIMING BYTE.
         MOVAH    /MOVE THE MSBY TO A.
         ORAL     /OR IT WITH THE LSBY.
         JNZ      /THE TIMING BYTE IS NONZERO, SO
         WAIT     /EXECUTE THE DCXH INSTRUCTION
         0        /AGAIN.
         POPH     /THE RESULT IS ZERO, POP H AND L OFF OF THE STACK,
         RET      /THEN RETURN.
```

When the 8080 returns from the HALF delay subroutine, the bit being received is again tested. If the data bit is still 0, then a genuine start bit is being detected. If the bit has returned to a logic 1, then the logic 0 being received was probably caused by noise. If this is the case, the 8080 must not try to interpret this noise as a serial character, so it jumps back to RCVR. If the start bit is still present one-half of a bit time after it was first detected, then the 8080 loads register pair B with 000 010 (0008). This instruction does two things. First, it clears register B. This is important since the B register will be used to temporarily store the character which will be received, one bit at a time. Second, it loads the bit count into the C register. In this example, the C register is loaded with decimal 8, so that an eight-bit character will be received. This is the number of data bits in the character to be received; it is not affected by the number of stop bits to be received. At NXTBIT, the DELAY subroutine is called. Why is this done?

When the 8080 executes the call to DELAY at NXTBIT, the start bit has already been transmitted to the 8080 for one-half of a bit time. By now waiting for one complete bit time, *the 8080 can sample the data input line in the middle of the next bit.* If the data stream is sampled in this manner, errors or differences in clock rates will be minimized. In our example, this means that the transmission speed can be slightly higher or lower than 110 bits per second, and the 8080 will still be able to correctly receive the data word. As an example, information can actually be transmitted to the 8080 at rates between 106 and 114 bits per second, even though the 8080 has been programmed to receive data at the rate of 110 bits per second. We have added arrows to the previous diagram of a serially transmitted character to indicate where the 8080 actually tests data at the one-bit input port. We have not drawn any arrows on the diagram before the start bit, simply because the 8080 will test the input port again and again, very quickly. If we drew arrows, they would be drawn one on top of another. This new diagram can be seen in Fig. 1-6.

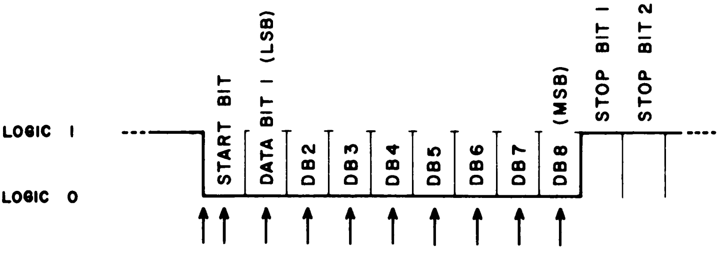


Fig. 1-6. The time relationship between the serial data and sampling.

The remaining bits are tested in a similar way to make up the eight-bit data word. After the DELAY subroutine is called, the data is input. Only the single data bit from the input port is saved in the A register (bit D_0) after the ANI instruction is executed. The content of register B is then added to the 0 or 1 contained in register A. The result of the addition is then rotated to the right once and then saved back in the B register. Since this rotation is performed by an RRC instruction, the content of the carry is not rotated into the A register. After the value is saved in the B register, the bit count in the C register is decremented by 1. If the data from the input port has not been input eight times, the count is nonzero, so the JNZ to NXTBIT is executed. After the eight data bits have been received, the 8080 calls the DELAY subroutine twice, so that the 8080 "receives" the two stop bits, although they are not input and checked by the 8080.

What has to be changed in the RCVR subroutine so that the 8080 microcomputer can receive data at the rate of 600 bits per second? The only section of the subroutine that has to be changed is the time delay subroutine HALF. Remember, this subroutine generates a time delay equal to one-half of a bit time. Therefore, a new timing byte can be calculated as follows:

$$\frac{1 \text{ second}}{600 \text{ bits}} = 1.66 \text{ ms} = \text{bit time}$$

$$\frac{1.66 \text{ ms}}{2} = 833\ \mu\text{s} = \text{one-half bit time}$$

The DCXH, MOVAH, ORAL, and JNZ instructions in the HALF subroutine require 24 cycles to be executed. If the 8080 has a cycle time of 500 ns, these instructions require 12 μs to be executed.

$$\frac{833\ \mu\text{s}}{12\ \mu\text{s}} = 69_{10} \text{ executions of the loop}$$

This means that the 16-bit timing byte loaded into reigster pair H must be equal to 69_{10}. A timing byte of 000 105 (0045) can be used. This number would have to be saved in memory just after the op code for the LXIH instruction in the HALF subroutine. *This timing byte assumes that the 8080 has a cycle time of 500 ns (2 MHz).* The timing bytes for a number of other bit rates are listed in Table 1-7.

Can the RCVR subroutine be used to *reliably* receive asynchronous serial characters at the rate of 19,200 bits per second? No, it cannot. The bit time for this data rate is 52.08 μs (1 bit/19,200 bits/s). This means that the HALF time delay subroutine must generate a delay of one-half of this, or 26.04 μs. If register pair H is loaded with 000 002 (0002), a delay of 24 μs will be produced by

Table 1-7. Timing Bytes for Different Reception Rates (500 ns Cycle Time; One-Half Bit Times)

Reception Rate (bits/s)	Decimal	Octal	Hexadecimal
110	378	001 172	017A
300	138	000 212	008A
600	69	000 105	0045
1,200	34	000 042	0022
2,400	17	000 021	0011
4,800	8	000 010	0008
9,600	4	000 004	0004
19,200	2	000 002	0002

the time delay loop within the HALF subroutine. However, an additional 20.5 μs is also required to execute the PUSHH, LXIH, POPH, and RET instructions. This means that a time delay of 44.5 μs is really generated by the HALF subroutine.

Is this additional delay ("overhead instructions") important when characters are received at the rate of 110 bits per second? No. For this data rate, a delay of 4.54 ms (4545 μs) must be produced. The additional delay of 20.5 μs produces an error of only 0.45%.

$$\frac{20.5\ \mu s}{4545\ \mu s} \times 100\% = 0.45\% \text{ error}$$

Unfortunately, as the data rates become higher, the error due to the "overhead instructions" becomes greater. The errors generated by the HALF subroutine are compiled in Table 1-8.

Table 1-8. Errors in the HALF Subroutine at Different Bit Rates (500 ns Cycle Time)

Bit Rate (bits/s)	Error
110	0.45%
300	1.23%
600	2.46%
1,200	4.92%
2,400	9.84%
4,800	19.68%
9,600	39.36%
19,200	78.72%

At what point will the 8080 not be able to reliably receive data? Our own opinion is that 1200 bits per second is the upper limit for this software. Remember, the errors in Table 1-8 are *per bit.* Therefore, for a start bit and an eight-bit data word, the total error at 1200 bits per second is 44.28%. If a higher reception rate is required, then the software will have to be *tuned up,* so that the time required to execute *all* of the instructions in the RCVR subroutine, including the CALL and RET instructions, is taken into account. However, this is beyond the scope of our present discussion.

Can the receiver subroutine be modified so that six-bit data words are received? This means that one start bit, a six-bit data word, and one or more stop bits will be transmitted to the 8080. The required modification is very simple. Just before NXTBIT there is an LXIB instruction. By changing the second byte of this instruction (the number that is loaded into the C register), the number of bits per word that the 8080 will detect can be changed. To receive a six-bit word, the number would be changed from 010 to 006 (08 to 06).

Suppose you change this data byte from 010 to 020 (08 to 10). What will happen? If we assume that a 16-bit data word is being transmitted to the 8080, it will receive the 16-bit data word. However, since an eight-bit register, the B register, is used to store the character as it is received, only the last eight bits of the word will be saved. Would these eight bits of information represent the MSBY or LSBY of the 16-bit word that was transmitted? The eight bits of information would be the MSBY of the transmitted data word. Finally, the RCVR subroutine can be modified to receive data words that are from one bit to eight bits in length. This can be done by changing the second byte of the LXIB instruction.

The RCVR subroutine (Example 1-6) changes the content of the A, B, and C registers. Are there any instructions that can be executed so that only the A register is altered? Yes, a PUSHB instruction could be executed at the beginning of the RCVR subroutine sometime before the LXIB instruction is executed. A MOVAB and POPB instruction sequence would then be executed just before the RET instruction. The MOVAB is required to move the received data word from the B register to the A register before the B and C registers are restored by the POPB instruction.

One of the most serious limitations associated with all of the software-based serialization methods is the fact that the 8080 cannot perform *any* other tasks while it is receiving or transmitting data. For instance, if a hardware UART chip is used with the 8080, the software can simply test the state of either the transmitter or receiver flag, or both. If the flags are not in the desired state, the 8080 can then perform other tasks for a few milliseconds. At the end of this time, the 8080 can again check the state of the flags. If they are in the desired state, the 8080 can input data from the receiver or output data to the transmitter. Once this is done, the 8080 can perform other tasks because the UART automatically transmits the data or receives the data without requiring further computer/UART interaction.

If a software-based serialization method is used by the 8080, then it will be executing either the transmitter or receiver software.

Once the 8080 begins the actual transmission or reception of a data word, it can perform no other tasks, simply because the 8080 must accurately time the period between each bit of information. This means that the 8080 is *I/O bound*, or it is dedicated to communicating with either the peripheral that it is sending data to, or receiving data from. No other operations can be performed by the 8080 while it is communicating with the peripheral device by means of a software-based serialization technique. A final comparison of UARTs and software-serialization techniques is shown in Table 1-9. The choice of using either a software scheme or a UART chip is yours. Both methods have their advantages and disadvantages.

Table 1-9. A Comparison of Hardware- and Software-Based UARTs

Feature Compared	Software	Hardware
Maximum Speed (bits/s)	1200 (Approx.)	DC to 50,000
Hardware Cost ($)	1–3	20
Software (No. of Memory Locations)	100	20
Power Supplies*	+5	+5, −12
Error Checking?	Maybe	Yes
I/O Bound?	Yes	No
Reliability?	Depends on Speed	Excellent
Bit Rate Determined By:	CPU Speed	External Clock
Receive and Transmit Simultaneously?	No	Yes
Communicate With Many Asynchronous Devices Simultaneously?	No	Yes

* Some UARTs are available that require only a single +5-volt power supply; however, they are more expensive than the UARTs that require two power supplies.

THE 8085 AND UART'S

Hardware UARTs can be used with the 8085 in exactly the same manner as with 8080-based microcomputers. The interfaces can be exactly the same and the software instructions that are used to communicate with the UART can be exactly the same. However, two new 8085 instructions are designed to make software serialization methods easier to implement. Improvements in the 40-pin 8085 integrated circuit also mean that fewer integrated circuits are required for the software-based serialization method.

Two of the pins on the 8085 integrated circuit are dedicated to serial communications. These two pins have been assigned the names *serial input data* (SID, pin 5) and *serial output data* (SOD, pin 4). Two new 8085 instructions are used to either input the data on the SID pin into bit D_7 of the A register, or to output bit D_7

Table 1-10. The Op Codes for the Single-Byte RIM and SIM Instructions

Op Code		Mnemonic	Operation
Octal	Hex		
040	20	RIM	Read the state of the SID (serial input data) pin into bit D_7 of the A register*.
060	30	SIM	Transfer the content of bit D_7 to the SOD (serial output data) pin if bit D_6 of the A register is a logic 1*.

* Both the RIM and SIM instructions cause other operations to take place between the A register of the 8085 and some of the internal logic of the 8085. This will be discussed in Chapter 3.

of the A register to the SOD pin. The RIM instruction reads the state of the SID pin into bit D_7 of the A register, and the SIM instruction transfers the state of bit D_7 of the A register to the SOD pin. As in previous software and hardware examples, additional circuitry will be required to convert the TTL-compatible levels of the SID and SOD pins to the 20 mA or RS-232C required by the teletypewriter or crt with which the 8085 is communicating. The op codes for the RIM and SIM instructions are shown in Table 1-10.

A software-based transmitter subroutine is listed in Example 1-7. It is slightly longer and more complex than the previous software-based serialization subroutine (Example 1-4). *This subroutine will only work on an 8085-based microcomputer.* The subroutine must be called with the data word, which is to be transmitted to the peripheral device, in the A register.

When the TRANS subroutine is called, register pairs H and B are saved on the stack. Register pair H is used in the DELAY subroutine to store a timing byte, and register pair B is used to store the bit count and the data word that is being transmitted. After the PUSHB instruction is executed, the data word in the A register is moved to the B register, and the C register is loaded with the number of *data bits* to be transmitted. By changing the value that is loaded into the C register, the 8085 can be programmed to transmit data words that contain from one to eight data bits. The MVIA instruction then sets bit D_7 to 0, bit D_6 to 1, and the remainder of the data bits in register A to 0. Bit D_7 represents the start bit, which must be transferred to the SOD pin before any data bits are transmitted. *Bit D_6 of the A register must be a logic 1 for the content of bit D_7 to be transferred to the SOD pin when the 8085 executes the SIM instruction.* The SIM instruction just before NXTBIT then transfers bit D_7 of the A register to the SOD pin because bit D_6 is a logic 1. The content of the A register is not changed by the SIM instruction.

Once the start bit has been transferred to the SOD pin, the 8085 calls the DELAY subroutine at NXTBIT. This causes a time delay of one bit time to be generated. When the 8085 returns from the DELAY subroutine, the eight-bit data word that is to be transmitted to the peripheral device is moved from the B register to the A register. The RRC instruction then rotates bit D_0 into bit D_7 so that it may be output by the SIM instruction. The rotated word is then saved in the B register. The ANI instruction sets all bits in the A register to 0, except bit D_7. It may appear that this instruction is unnecessary, simply because the SIM instruction transfers only bit D_7 to the SOD pin. However, when the SIM instruction is executed, bits D_0 through D_4 cause other actions to occur within the 8085 integrated circuit. This will be discussed in the next chapter.

By setting these bits to 0, only the state of the SOD pin is affected by the SIM instruction. The ADI instruction sets bit D_6 of the A register to a logic 1. An ORI instruction could have also been used here in the subroutine. Bit D_6 of the A register must be a logic 1 so that the state of bit D_7 is transferred to the SOD pin when the SIM instruction is executed. If bit D_6 is a logic 0, then the SIM instruction will not affect the state of the SOD pin.

After the ADI instruction is executed, the SIM instruction transfers bit D_7 to the SOD pin. The bit count contained in the C register is then decremented. If the result of this action is nonzero, the JNZ to NXTBIT is executed. At NXTBIT, the DELAY subroutine is called, so that the bit of information just output to the SOD pin will remain there for one bit time. Each bit of information is then transmitted at regular DELAY intervals, thus generating the serial flow of data.

When all of the data bits have been transmitted, as indicated by the content of the C register being decremented to 0, the JNZ to NXTBIT is not executed. Instead, the DELAY subroutine is called, so that the last bit of information is transmitted for a complete bit time. The A register is then loaded with 300 (C0), and another SIM instruction is executed. This sets the SOD pin to a logic 1. This is the beginning of one or more stop bits. In Example 1-7, the DELAY subroutine is called twice, so that two stop bits are transmitted to the serial device. When the transmission has been completed, register pairs B and H are popped off of the stack and the 8085 returns from the TRANS subroutine.

Example 1-7: A Software-Based Asynchronous Serial Transmitter Subroutine for the 8085 Only

```
/THIS SUBROUTINE, WHICH SHOULD ONLY BE EXECUTED
/ON AN 8085, TRANSMITS EIGHT-BIT ASYNCHRONOUS
/CHARACTERS USING THE SIM INSTRUCTION.

TRANS,  PUSHH   /SAVE REGISTER PAIR H ON THE STACK.
        PUSHB   /SAVE REGISTER PAIR B ON THE STACK.
```

```
          MOVBA    /SAVE THE CHARACTER IN B.
          MVIC     /LOAD THE C REGISTER WITH THE
          010      /NUMBER OF DATA BITS PER CHARACTER.
          MVIA     /LOAD A WITH 01000000, TO SEND A ZERO
          100      /START BIT (D6 = 1 FOR THE SIM INSTRUCTION).
          SIM      /OUTPUT THE ZERO TO THE SOD PIN.
NXTBIT,   CALL     /CALL THE DELAY SUBROUTINE THAT
          DELAY    /PRODUCES A DELAY OF ONE BIT TIME.
          0
          MOVAB    /GET THE CHARACTER BEING TRANSMITTED.
          RRC      /ROTATE D0 TO D7 AND THE CARRY.
          MOVBA    /SAVE THE CHARACTER BACK IN B.
          ANI      /SAVE ONLY D7 OF THE WORD BEING TRANSMITTED
          200      /(200 = HEX 80).
          ADI      /ADD 01000000 SO THAT THE SIM INSTRUCTION
          100      /TRANSMITS THE BIT OF DATA (100 = HEX 40).
          SIM      /SEND BIT D7 TO THE SOD PIN OF THE 8085.
          DCRC     /DECREMENT THE BIT COUNT.
          JNZ      /IF THE COUNT IS NONZERO,
          NXTBIT   /JUMP BACK AND TRANSMIT ANOTHER
          0        /DATA BIT.
          CALL     /NOW TRANSMIT THE LAST DATA
          DELAY    /BIT FOR ONE BIT TIME.
          0
          MVIA     /NOW SEND TWO STOP BITS
          300      /(300 = HEX C0) 11000000.
          SIM      /SEND A ONE TO THE SOD PIN.
          CALL     /CALL THE DELAY SUBROUTINE
          DELAY    /FOR A SINGLE BIT TIME.
          0        /(THIS IS ONE STOP BIT)
          CALL     /CALL THE DELAY SUBROUTINE A SECOND
          DELAY    /TIME TO TRANSMIT A SECOND STOP BIT.
          0
          POPB     /POP REGISTER PAIR B OFF OF THE STACK.
          POPH     /POP REGISTER PAIR H OFF OF THE STACK.
          RET      /THEN RETURN FROM THE SUBROUTINE.

DELAY,    PUSHPSW  /SAVE THE FLAGS AND A ON THE STACK.
          LXIH     /LOAD REGISTER PAIR H WITH
          365      /A 16-BIT TIMING BYTE.
          002
WAIT,     DCXH     /DECREMENT THE TIMING BYTE BY ONE.
          MOVAH    /MOVE THE MSBY INTO A.
          ORAL     /OR IT WITH THE LSBY.
          JNZ      /IF THE 16-BIT NUMBER IS NON-
          WAIT     /ZERO, THEN JNZ TO THE DCXH
          0        /INSTRUCTION.
          POPPSW   /POP A AND THE FLAGS OFF OF THE STACK.
          RET      /THEN RETURN FROM THE DELAY SUBROUTINE.
```

How can the TRANS subroutine be modified so that only one stop bit is transmitted to the peripheral device? This is done simply by removing one of the calls to the DELAY subroutine at the end of the subroutine, just before the POPB instruction.

How can two memory locations be saved by changing Example 1-7? Simply eliminate the PUSHPSW and POPPSW instructions in the DELAY subroutine. In the previous software-based asynchronous serial transmitter subroutine, these instructions were necessary because register A contained the data word that was being transmitted to the peripheral device. However, in Example 1-7, the data word being transmitted is saved in the B register, so these two instructions serve no useful purpose.

As you can see, the transmitter subroutine for the 8085 is longer than the software-based asynchronous serial transmitter subroutine that can be executed by the 8080. However, for the software transmission scheme to be implemented on an 8080-based microcomputer, a one-bit input port and a one-bit output port must be constructed. Because of the improvements in the 8085 microprocessor integrated circuit, these two ports are already contained on the 8085 chip.

8085-BASED, SOFTWARE-BASED, ASYNCHRONOUS SERIAL RECEIVER SOFTWARE

The differences between the two receiver subroutines are not as great. In fact, there are only three different instructions. First, RIM instructions are used to read the state of the SID pin into bit D_7 of the A register. Since the RIM instruction does not read the bit of information into bit D_0, the ANI mask must be changed from 001 to 200 (01 to 80). Finally, since the first data bit that will be received is the LSB, the data bits must be rotated to the right as they are received. Since *every* data bit is rotated to the right at least once, including the MSB of the received data word, an RLC instruction must be added to the end of the subroutine (Example 1-8). As you can see, this subroutine is very similar to the previous software-based asynchronous serial receiver subroutine (Example 1-6).

Example 1-8: A Software-Based Asynchronous Serial Receiver Subroutine for the 8085 Only

```
/THIS 8085 SUBROUTINE USES THE SID PIN AND THE
/RIM INSTRUCTION TO RECEIVE EIGHT-BIT ASYNCHRONOUS
/SERIAL CHARACTERS. THE RECEIVED CHARACTER IS IN
/A WHEN CONTROL RETURNS FROM THE SUBROUTINE.

RCVR,    PUSHB    /SAVE REGISTER PAIR B ON THE STACK.
RCVR1,   RIM      /READ THE SID PIN INTO D7 OF A.
         ANI      /SAVE ONLY THE DATA BIT IN D7
         200      /(200 = HEX 80).
         JNZ      /NOTHING IS BEING TRANSMITTED, NOT
         RCVR1    /EVEN A START BIT, SO KEEP WAITING
```

```
          0         /FOR THE START BIT.
          CALL      /A ZERO WAS DETECTED, SO WAIT FOR ONE-
          HALF      /HALF A BIT TIME.
          0
          RIM       /READ THE SID PIN AGAIN TO ENSURE
          ANI       /THAT A START BIT WAS DETECTED
          200       /(200 = HEX 80).
          JNZ       /IT WAS NOISE, SO JUMP BACK UP TO
          RCVR1     /THE BEGINNING OF THE SUBROUTINE.
          0
          LXIB      /IT WAS A VALID START BIT, SET B TO
          010       /000 FOR TEMPORARY STORAGE AND C TO
          000       /010 BECAUSE IT IS THE BIT COUNTER.
NEXTBIT,  CALL      /NOW WAIT FOR ONE BIT TIME BEFORE
          DELAY     /A SINGLE BIT IS INPUT INTO BIT
          0         /D7 OF THE A REGISTER.
          RIM       /NOW READ THE SID PIN AND SAVE
          ANI       /ONLY THE SINGLE DATA BIT
          200       /(200 = HEX 80).
          ADDB      /ADD THE TEMPORARY WORD TO THE BIT.
          RRC       /ROTATE IT TO THE RIGHT ONE POSITION.
          MOVBA     /THEN SAVE THE RESULT IN B.
          DCRC      /DECREMENT THE BIT COUNTER.
          JNZ       /IT IS NONZERO, SO GET ANOTHER BIT
          NXTBIT    /BUT FIRST WAIT AN ENTIRE BIT TIME
          0         /BEFORE INPUTTING THE DATA BIT.
          CALL      /NOW WAIT FOR TWO BIT TIMES FOR
          DELAY     /THE STOP BITS TO "GO BY."
          0
          CALL
          DELAY
          0
          MOVAB     /MOVE THE RECEIVED CHARACTER TO A.
          RLC       /ROTATE IT ONCE TO THE LEFT
          RET       /AND RETURN WITH THE CHARACTER IN A.

DELAY,    CALL      /TO GENERATE A DELAY OF ONE
          HALF      /BIT TIME, EXECUTE THE HALF SUB-
          0         /ROUTINE TWICE.
HALF,     PUSHH     /SAVE H AND L ON THE STACK.
          LXIH      /LOAD REGISTER PAIR H WITH A 16-BIT
          172       /TIMING BYTE WHICH IS ONE-HALF
          001       /THE BIT RATE.
WAIT,     DCXH      /DECREMENT THE TIMING BYTE.
          MOVAH     /MOVE THE MSBY TO A.
          ORAL      /OR IT WITH THE LSBY.
          JNZ       /THE TIMING BYTE IS NONZERO, SO
          WAIT      /EXECUTE THE DCXH INSTRUCTION
          0         /AGAIN.
          POPH      /THE RESULT IS ZERO, POP H AND L OFF OF THE STACK.
          RET       /THEN RETURN.
```

Is there any advantage in using an 8085 in a microcomputer system rather than an 8080? If the microcomputer must communicate with a single asynchronous serial peripheral device, then the use of a software-based set of transmitter and receiver subroutines with the 8085 will probably save a few integrated circuits. If an 8080 is used, then some circuitry will be required to build a one-bit input port and a one-bit output port. If the microcomputer must communicate with a number of different asynchronous serial peripheral devices, the 8085 has few advantages over the 8080, simply because the communications task will probably be performed by UART or USART devices.

In Table 1-9, we listed a number of characteristics of hardware- and software-based UARTs. However, one point from this table should be emphasized. The microcomputer cannot transmit and receive data simultaneously using a software-based asynchronous serial scheme alone. Suppose that the program in Example 1-9 is executed using a software-based scheme. The data is transmitted to and received from a 10-character-per-second teletypewriter. What is the maximum number of characters that can be entered on the keyboard and printed on the printer in one second? It does not matter if the program is executed on an 8080- or 8085-based microcomputer.

Example 1-9: A Simple Software-Based Reception and Transmission Test Program

```
/THIS IS A SIMPLE SOFTWARE-BASED SERIALIZATION
/AND RECEPTION TEST PROGRAM.

START,  LXISP   /LOAD THE STACK POINTER WITH
        STACK   /A R/W MEMORY ADDRESS.
        0
LOOP,   CALL    /GET A CHARACTER FROM THE
        RCVR    /KEYBOARD.
        0
        CALL    /THEN TRANSMIT THE CHARACTER
        TRANS   /TO THE PRINTER.
        0
        JMP     /THEN DO IT AGAIN.
        LOOP
        0
```

Using the software listed in Example 1-9, only *five* characters can be received and transmitted in one second. The reason for this should be obvious: the microcomputer needs 100 ms to receive a character and 100 ms to transmit the same character back to the printer. Therefore, 200 ms are required to receive and transmit a single character. This may be an important point to remember if you choose to apply the software-based serialization method to communications problems. This problem becomes less important as bit rates increase to 600 or 1200 bits per second.

HARDWARE DEVICES—USART'S VERSUS UART'S

Since the *U*niveral *S*ynchronous/*A*synchronous *R*eceiver/*T*ransmitter (USART) is a newer device than the UART, it has all of the features of the UART and some additional features as well. The USART is usually packaged as a 28-pin integrated circuit. Unlike the UART, however, the USART has been designed primarily for use with microcomputers. The pin configuration and block diagram for the Intel 8251 USART can be seen in Fig. 1-7.

There are eight data lines (D_0 through D_7) that the 8080 uses to communicate with the USART. These data lines are used to output data to the transmitter in the USART and to input data from the receiver in the USART. The 8080 can also output a *mode word* and a *command word* to the USART. The mode word programs the parity of the USART, the number of stop bits, and a *baud rate factor.*

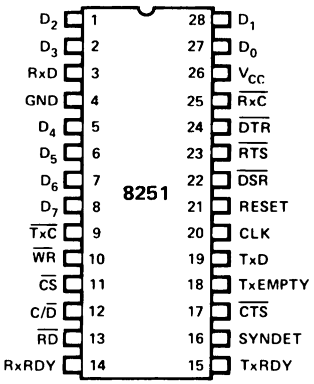
With a UART, external jumpers are used to program the parity and number of stop bits. The ability to program the USART with a baud rate factor using software is a feature not directly available on a standard UART. The baud rate factor is a divisor that is applied to both the transmitter and receiver clocks for a specific transmission or reception rate. The three baud rate factors that can be selected are ×1, ×16, and ×64. This means that the required frequency of the clock applied to the transmitter and receiver sections of the USART (pins 9 and 25) is really determined by the bit rate required and the baud rate factor programmed into the USART. The baud rate factor is part of the mode word that is programmed into the USART. The different baud rate factors and clock frequencies are summarized in Table 1-11 for the "usual" bit rates.

If a bit rate of 1200 bits per second is required, then a frequency of 1200 Hz, 19.2 kHz, or 76.8 kHz could be applied to the clock

Table 1-11. The Clock Frequencies Required for Different Bit Rates and Baud Rate Factors

	Clock Frequency Required for Baud Rate Factors		
Bit Rate	**×1**	**×16**	**×64**
110	110 Hz	1.76 kHz	7.04 kHz
150	150 Hz	2.4 kHz	9.6 kHz
300	300 Hz	4.8 kHz	19.2 kHz
600	600 Hz	9.6 kHz	38.4 kHz
1,200	1,200 Hz	19.2 kHz	76.8 kHz
2,400	2,400 Hz	38.4 kHz	153.6 kHz
4,800	4,800 Hz	76.8 kHz	307.2 kHz
9,600	9,600 Hz	153.6 kHz	614.4 kHz
19,200	19,200 Hz	307.2 kHz	1,228.8 kHz

PIN CONFIGURATION

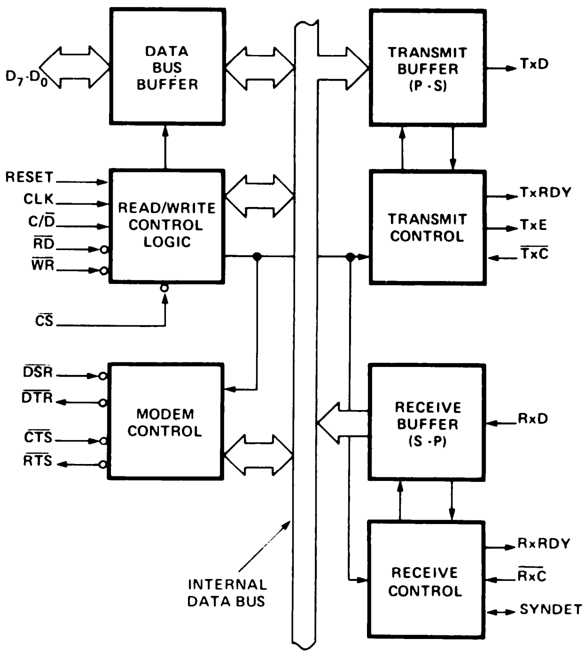


D2
D3
RxD
GND
D4
D5
D6
D7
$\overline{\text{TxC}}$
$\overline{\text{WR}}$
$\overline{\text{CS}}$
C/$\overline{\text{D}}$
$\overline{\text{RD}}$
RxRDY
1
2
3
4
5
6
7
8
9
10
11
12
13
14
8251
28
27
26
25
24
23
22
21
20
19
18
17
16
15
D1
D0
VCC
$\overline{\text{RxC}}$
$\overline{\text{DTR}}$
$\overline{\text{RTS}}$
$\overline{\text{DSR}}$
RESET
CLK
TxD
TxEMPTY
$\overline{\text{CTS}}$
SYNDET
TxRDY

Pin Name	Pin Function
D_7-D_0	Data Bus (8 bits)
$C/\overline{D}$	Control or Data is to be Written or Read
$\overline{\text{RD}}$	Read Data Command
$\overline{\text{WR}}$	Write Data or Control Command
$\overline{\text{CS}}$	Chip Enable
CLK	Clock Pulse (TTL)
RESET	Reset
$\overline{\text{TxC}}$	Transmitter Clock
TxD	Transmitter Data
$\overline{\text{RxC}}$	Receiver Clock
RxD	Receiver Data
RxRDY	Receiver Ready (has character for 8080)
TxRDY	Transmitter Ready (ready for char. from 8080)

Pin Name	Pin Function
$\overline{\text{DSR}}$	Data Set Ready
$\overline{\text{DTR}}$	Data Terminal Ready
SYNDET	Sync Detect
$\overline{\text{RTS}}$	Request to Send Data
$\overline{\text{CTS}}$	Clear to Send Data
TxE	Transmitter Empty
V_{CC}	+5 Volt Supply
GND	Ground

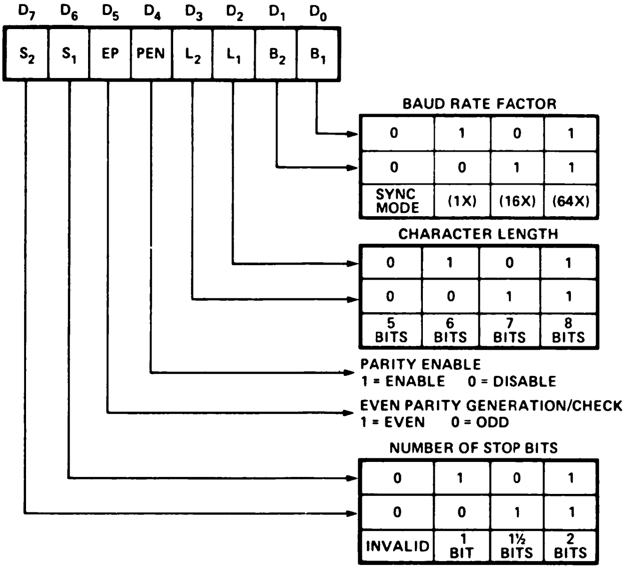
BLOCK DIAGRAM



D7-D0
DATA BUS BUFFER
RESET
CLK
C/$\overline{\text{D}}$
$\overline{\text{RD}}$
$\overline{\text{WR}}$
$\overline{\text{CS}}$
READ/WRITE CONTROL LOGIC
$\overline{\text{DSR}}$
$\overline{\text{DTR}}$
$\overline{\text{CTS}}$
$\overline{\text{RTS}}$
MODEM CONTROL
INTERNAL DATA BUS
TRANSMIT BUFFER (P-S)
TxD
TRANSMIT CONTROL
TxRDY
TxE
$\overline{\text{TxC}}$
RECEIVE BUFFER (S-P)
RxD
RECEIVE CONTROL
RxRDY
$\overline{\text{RxC}}$
SYNDET

Courtesy Intel Corp.

Fig. 1-7. The pin configuration and block diagram for the Intel 8251 USART.


D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
S_2 S_1 EP PEN L_2 L_1 B_2 B_1
BAUD RATE FACTOR
0 1 0 1
0 0 1 1
SYNC MODE (1X) (16X) (64X)
CHARACTER LENGTH
0 1 0 1
0 0 1 1
5 BITS 6 BITS 7 BITS 8 BITS
PARITY ENABLE
1 = ENABLE 0 = DISABLE
EVEN PARITY GENERATION/CHECK
1 = EVEN 0 = ODD
NUMBER OF STOP BITS
0 1 0 1
0 0 1 1
INVALID 1 BIT 1½ BITS 2 BITS

Courtesy Intel Corp.

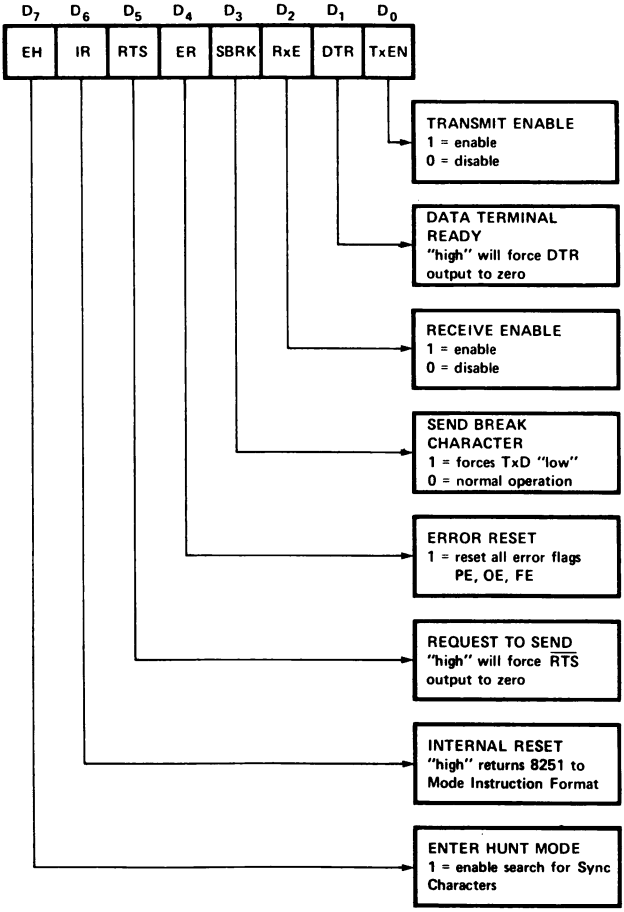
Fig. 1-8. The mode word format for the 8251 USART.

pins of the USART. The appropriate baud rate factors for these clock frequencies are ×1, ×16, and ×64.

The format of the eight-bit mode word, which must be loaded into the USART before the USART can be used for communications with a peripheral device, is shown in Fig. 1-8. After the mode word has been loaded into the USART, a command word must also be loaded into the USART. The command word is used to enable and disable both the transmitter and the receiver sections of the USART. The 8251 also has the ability of controlling a *modem,* so two bits within the command word may be used for modem control. Another bit of the command word is used to reset the parity error, overrun error, and framing error flags. These flags have the same function on the USART as they do on the UART. The format for the 8251 command word is shown in Fig. 1-9.

Note that bit D_6 of the command word can be used to reset the 8251 USART back to the mode word programming step. This feature is required because the mode word and command word are output to the *same output port in the USART!* When the USART is reset, either by applying a positive pulse to the RESET pin (pin 21) or by setting bit D_6 of the command word to a logic 1, the USART is reset so that a mode word can be output to it.

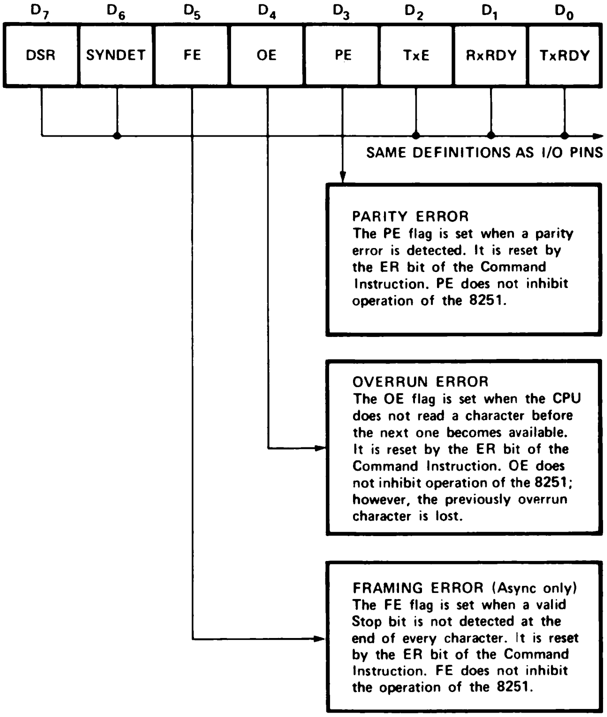
When the mode word is output to the USART, it is written into the mode word register. The 8251 then performs some internal switching, so that the next time a word is output to the control section of the USART, it will be written into the command word register. Only after the RESET pin of the USART has been pulsed, or when bit D_6 of the command word is a 1, should a new mode word be output to the 8251 USART. It is also important to remember that the mode word *must always* be followed by the command word. Once the 8251 has been programmed with a mode word *and* a command word, it can be used for either synchronous or asynchronous communications. Most USART users use the 8251 for asynchronous communications, so synchronous communications will not be discussed.


D7 D6 D5 D4 D3 D2 D1 D0
EH
IR
RTS
ER
SBRK
RxE
DTR
TxEN
TRANSMIT ENABLE
1 = enable
0 = disable
DATA TERMINAL READY
"high" will force DTR output to zero
RECEIVE ENABLE
1 = enable
0 = disable
SEND BREAK CHARACTER
1 = forces TxD "low"
0 = normal operation
ERROR RESET
1 = reset all error flags PE, OE, FE
REQUEST TO SEND
"high" will force RTS output to zero
INTERNAL RESET
"high" returns 8251 to Mode Instruction Format
ENTER HUNT MODE
1 = enable search for Sync Characters

Courtesy Intel Corp.

Fig. 1-9. The command word format for the 8251 USART.

During the transmission and reception of data words, the 8080 still has to be able to monitor the status of the transmitter and receiver so that it can determine when to output another data word to the transmitter or when to input a data word from the receiver. The USART *status word* contains two flags just for this purpose. It also contains three error flags and two additional flags that we will not discuss. An additional flag, the DSR (*data set ready*) flag, is used for modem control. The format of the status word is shown in Fig. 1-10.



PARITY ERROR
The PE flag is set when a parity error is detected. It is reset by the ER bit of the Command Instruction. PE does not inhibit operation of the 8251.

OVERRUN ERROR
The OE flag is set when the CPU does not read a character before the next one becomes available. It is reset by the ER bit of the Command Instruction. OE does not inhibit operation of the 8251; however, the previously overrun character is lost.

FRAMING ERROR (Async only)
The FE flag is set when a valid Stop bit is not detected at the end of every character. It is reset by the ER bit of the Command Instruction. FE does not inhibit the operation of the 8251.

Courtesy Intel Corp.

Fig. 1-10. The status word format for the 8251 USART.

As you know, the USART really contains two input devices and two output devices. Either the status word or a data word can be read from the USART, and a data word or the mode/command word can be output to it. When an eight-bit word is output to the mode/command section of the USART, the first word must always be the mode word. The command word must then be output to the USART. Any number of different mode and command words can be output to the USART, but the mode word must always be output first.

There are four control pins on the USART that actually control the flow of data between the USART and the 8080. These signals are *write* ($\overline{\text{WR}}$), *read* ($\overline{\text{RD}}$), *control/data* (C/$\overline{\text{D}}$), and *chip select* ($\overline{\text{CS}}$). The truth table for these four inputs of the 8251 is shown in Table 1-12. Based on this truth table, an interface between the USART and the 8080 can be readily designed (Fig. 1-11). The $\overline{\text{CS}}$ input of the 8251 (pin 11) will only go to a logic 0 when the following logic levels are present on A_1 through A_7 of the address bus (performed by a decoder):

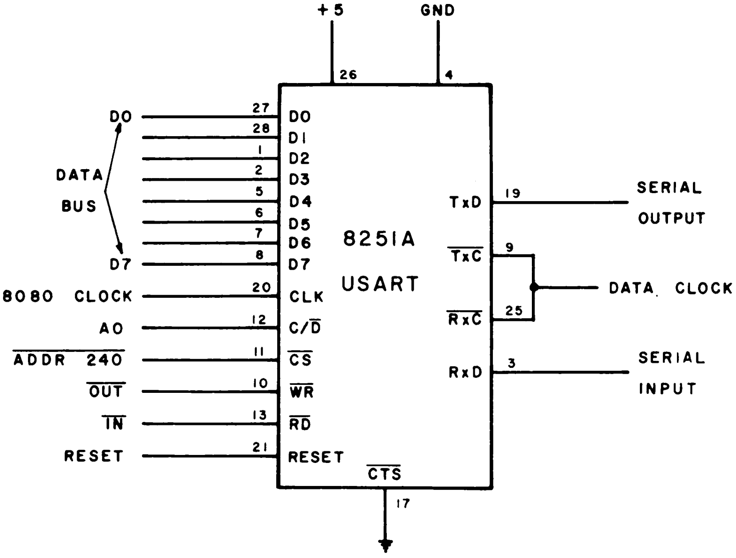

+5
GND
26
4
D0
DATA
BUS
D7
8080 CLOCK
A0
$\overline{\text{ADDR 240}}$
$\overline{\text{OUT}}$
$\overline{\text{IN}}$
RESET
27
28
1
2
5
6
7
8
20
12
11
10
13
21
D0
D1
D2
D3
D4
D5
D6
D7
CLK
C/$\overline{\text{D}}$
$\overline{\text{CS}}$
$\overline{\text{WR}}$
$\overline{\text{RD}}$
RESET
8251A
USART
TxD
$\overline{\text{TxC}}$
$\overline{\text{RxC}}$
RxD
$\overline{\text{CTS}}$
17
19
9
25
3
SERIAL
OUTPUT
DATA CLOCK
SERIAL
INPUT

Fig. 1-11. An accumulator I/O interface for the 8251 USART.

A_7	A_6	A_5	A_4	A_3	A_2	A_1
1	0	1	0	0	0	0

Table 1-12. Truth Table for the Control Signals of the 8251 USART

C/$\overline{\text{D}}$	$\overline{\text{RD}}$	$\overline{\text{WR}}$	$\overline{\text{CS}}$	Function
0	0	1	0	Read the receiver data.
0	1	0	0	Output data to the transmitter.
1	0	1	0	Read the status word.
1	1	0	0	Output the mode/command word.
X	X	X	1	The USART is not selected; the 8080 cannot communicate with it.
X = Don't care; a logic 1 or a logic 0.				

Based on the logic levels of these address lines, a truth table can be constructed so that the input and output port addresses for the USART can be easily determined (Table 1-13).

Table 1-13. The Input and Output Port Addresses for the USART

A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	Function
1	0	1	0	0	0	0	0	0	1	0	Read the receiver.
1	0	1	0	0	0	0	0	1	0	0	Load the transmitter.
1	0	1	0	0	0	0	1	0	1	0	Read the status.
1	0	1	0	0	0	0	1	1	0	0	Load the mode/command register.
X	X	X	X	X	X	X	X	X	X	1	USART is not selected.

X = Don't care; a logic 1 or a logic 0. (A_0 wired to $C/\overline{D}$.)

Since the read ($\overline{RD}$) and write ($\overline{WR}$) lines of the USART are wired to $\overline{IN}$ ($\overline{I/O\ R}$) and $\overline{OUT}$ ($\overline{I/O\ W}$), accumulator I/O instructions must be used to communicate with the USART. To input the receiver data, an IN 240 must be executed. To load the transmitter with a data word, the content of the A register would have to be output to output port 240. To output a mode or a command word to the USART, the content of the A register would have to be output to output port 241. To read the status of the transmitter and receiver within the USART, the 8080 would have to be programmed to input information into the A register from input port 241.

If an 8080 is equipped with a USART, the program listed in Example 1-10 might be executed. What has been programmed into the USART with these four instructions? In binary, the mode word is 11010111. This programs the USART for two stop bits and odd parity. Note that the odd parity is selected because the parity bit (D_5) is a logic 0 and the parity enable bit (D_4) is a logic 1, enabling the generation of parity. The mode word also programs the USART to transmit/receive six data bits per word and the baud rate factor selected is ×64. The 8080 then outputs a command word of 00010101; this simply resets all of the error flags and enables both the transmitter and receiver within the USART. Note that the mode word and the command word have both been output to the same output port.

Suppose that the USART is programmed with the two different mode words, 11101111 and 11001111. Will the 8251 USART perform any functions differently if one or the other mode word is used? It does not matter what the command word is. You can assume that it is the same for both mode words. The only difference between these two mode words is that one will attempt to program the USART for even parity and the other will attempt to program

Example 1-10: Programming the Mode and Command Registers of the USART

```
  •
  •
MVIA    /LOAD A WITH THE FOLLOWING DATA BYTE
327     /(HEXADECIMAL D7 OR BINARY 11010111).
OUT     /OUTPUT IT TO THE USART TO
241     /DETERMINE THE MODE CONDITIONS.
MVIA    /THEN THE COMMAND WORD HAS TO BE OUTPUT
025     /(HEXADECIMAL 15, BINARY 00010101).
OUT     /OUTPUT THIS COMMAND WORD TO THE
241     /USART.
  •
  •
```

the USART for odd parity. However, since bit D_4 of both these mode words is 0, *no* parity will be used by the USART.

The actual transmitter and receiver subroutines for the USART are very similar to the subroutines used with a UART. These two USART subroutines are listed in Example 1-11. In the UTRANS subroutine, the character to be output to the USART must be contained in the A register before the subroutine is called. Rather than save the character in the B register while the USART transmitter flag is being checked, the character is saved on the stack.

There are a few additional signals required by the USART for it to function. The *clear to send* ($\overline{\text{CTS}}$, pin 17) pin of the USART

Example 1-11: Accumulator I/O USART Receiver and Transmitter Subroutines

```
URCVR,   IN       /INPUT THE STATUS WORD OF THE USART
         241      /(HEXADECIMAL A1).
         ANI      /SAVE ONLY THE RECEIVER'S STATUS
         002      /BIT (RXRDY) IN A.
         JZ       /IF THE BIT IS 0, THEN NO CHARACTER
         URCVR    /HAS BEEN RECEIVED, SO KEEP WAITING.
         0
         IN       /A CHARACTER HAS BEEN RECEIVED, SO
         240      /INPUT IT INTO THE A REGISTER
         RET      /AND RETURN WITH IT IN A.

 UTRANS, PUSHPSW /SAVE THE CHARACTER IN A ON THE STACK.
 UTRAN1, IN       /INPUT THE STATUS WORD OF THE USART.
         241
         ANI      /AND SAVE ONLY THE TRANSMITTER'S STATUS
         001      /BIT (TXRDY).
         JZ       /IF THE FLAG IS 0, A CHARACTER IS BEING
         UTRAN1   /TRANSMITTED, SO WAIT FOR IT TO BE
         0        /DONE.
         POPPSW   /GET THE CHARACTER BACK IN A.
         OUT      /THEN OUTPUT IT TO THE USART.
         240
         RET
```

should be wired to ground. This enables the transmitter if bit D_0 of the command word is also a logic 0. There is also a *clock input* (CLK, pin 20) to the USART. This clock input is used for internal timing and gating by the USART. *The frequency of this clock is not related to the transmission or reception speed of the 8251 USART*. However, the frequency of this clock must be at least 30 times greater than the transmitter or receiver clock frequency if the USART is being used for synchronous communications. For asynchronous communications, this clock frequency must be at least four times greater than the bit rate of the USART. This clock can be derived from the TTL ϕ_2 clock of the 8224 clock generator used with the 8080, or from the CLK (OUT) pin of the 8085.

Like the UART, the USART requires separate TTL-compatible clock signals for the receiver and transmitter sections. These clock signals are applied to the $\overline{\mathrm{RxC}}$ pin for the receiver and to the $\overline{\mathrm{TxC}}$ pin for the transmitter. The clock rate may be one, 16, or 64 times the bit rate, as programmed by bits D_1 and D_0 in the mode control word. These clock signals may be derived from the crystal timebase of the computer or from a separate oscillator circuit, as applicable. Finally, the 8251 USART requires only a single +5-volt power supply.

MEMORY-MAPPED UART'S AND USART'S

There is no reason why UARTs and USARTs cannot be interfaced to an 8080 microcomputer using memory-mapped I/O techniques. Instead of using $\overline{\mathrm{IN}}$ ($\overline{\mathrm{I/O\ R}}$) and $\overline{\mathrm{OUT}}$ ($\overline{\mathrm{I/O\ W}}$), the signals $\overline{\mathrm{MEMR}}$ and $\overline{\mathrm{MEMW}}$ are used to control the operation of the USART. This also means that a 16-bit address must be decoded by the interface electronics so that *only* the USART is enabled when the appropriate 16-bit address is placed on the address bus by the 8080.

Because UARTs and USARTs use very similar interfaces, all of our software examples will be for USARTs. In the following subroutines, it has been assumed that the USART has been assigned the addresses 200 240 (80A0) and 200 241 (80A1).

The LXIH instruction in Example 1-12 loads register pair H with the 16-bit address assigned to the mode/command register of the USART. A mode word of 337 (DF) and then a command word of 025 (15) is written out to the mode/command register of the USART. Note that the memory address does not change for the command word. The mode word programs the USART for two stop bits, odd parity, eight-bit data words, and a ×64 baud rate factor. The command word resets the error flags and enables both the receiver and transmitter.

Example 1-12: Initialization Instructions for a Memory-Mapped I/O USART

```
/THESE USART INITIALIZATION INSTRUCTIONS
/ARE USED TO INITIALIZE A MEMORY-MAPPED
/I/O USART.

        •
        •
        LXIH    /LOAD REGISTER PAIR H WITH THE
        241     /16-BIT ADDRESS FOR THE USART.
        200
        MVIM    /WRITE THE MODE WORD OUT
        337     /TO THE USART.
        MVIM    /AND THEN WRITE THE COMMAND WORD
        025     /OUT TO THE USART.
        •
        •
```

A receiver subroutine for use with a memory-mapped I/O USART is listed in Example 1-13. In this subroutine, register pair H is used to store the 16-bit memory address for the USART. Because of this, the first instruction in the subroutine saves register pair H on the stack. Register pair H is then loaded with the 16-bit address of the USART, 200 241 (80A1). This is the address for the status word within the USART. The A register is then loaded with the receiver flag mask of 002 (02). The status word addressed by register pair H is then ANDed with the content of the A register. If the receiver flag is a logic 0, the JZ to RWAIT is executed. When a character is finally received by the USART, the receiver flag will be set to a logic 1.

When the flag is a logic 1, the 8080 does not jump back to RWAIT. Instead, the DCXH instruction is executed, which decre-

Example 1-13: A Memory-Mapped I/O USART Receiver Subroutine

```
/THIS RECEIVER SUBROUTINE COMMUNICATES
/WITH A MEMORY-MAPPED I/O USART.

MRCVR,  PUSHH   /SAVE REGISTER PAIR H ON THE STACK.
        LXIH    /THEN LOAD REGISTER PAIR H WITH
        241     /THE 16-BIT ADDRESS ASSIGNED TO
        200     /THE USART.
RWAIT,  MVIA    /LOAD A WITH THE FLAG MASK.
        002
        ANAM    /AND A WITH THE STATUS WORD.
        JZ      /BIT D1 OF THE WORD IS ZERO
        RWAIT   /SO WAIT FOR THE FLAG TO BE
        0       /A LOGIC ONE.
        DCXH    /THEN DECREMENT THE I/O ADDRESS.
        MOVAM   /READ THE USART CHARACTER INTO A.
        POPH    /POP REGISTER PAIR H OFF OF THE STACK.
        RET     /RETURN WITH THE CHARACTER IN A.
```

ments the content of register pair H from 200 241 to 200 240 (80A1 to 80A0), which is the memory address for both the receiver and transmitter within the USART. The MOVAM instruction transfers the content of the USART receiver into the A register of the 8080, and then register pair H is popped off of the stack. The 8080 returns from this subroutine with the eight-bit character received in the A register. Note that a MOVDM or MOVBM instruction, or a number of other memory reference instructions, could have been used to read the receiver character into the 8080.

A subroutine that writes data to the USART transmitter is just as simple (Example 1-14). This subroutine also uses register pair H to store the 16-bit address of the USART. Therefore, when this subroutine is called, register pair H and the PSW are pushed onto the stack. Register A contains the character to be written out to the transmitter. At TWAIT, the A register is loaded with the mask for the transmitter flag. The ANAM instruction then sets or clears the 8080 zero flag, based on the state of the transmitter flag. If the flag is 0, the 8080 jumps back to TWAIT.

When the flag is a logic 1, the USART is ready to transmit another character. Therefore, the DCXH instruction decrements the content of register pair H so that it will address the transmitter in the USART. The POPPSW instruction pops A and the flags off of the stack. This restores the character to be written out to the USART. This data word is then output to the USART when the MOVMA instruction is executed. Register pair H is then popped off of the stack and the 8080 returns to the section of the program that called MTRANS.

Example 1-14: A Memory-Mapped I/O USART Transmitter Subroutine

```
/THIS TRANSMITTER SUBROUTINE COMMUNICATES
/WITH A MEMORY-MAPPED I/O USART.

MTRANS, PUSHH   /SAVE REGISTER PAIR H ON THE STACK.
        PUSHPSW /SAVE A AND THE FLAGS ON THE STACK.
        LXIH    /LOAD REGISTER PAIR H WITH THE
        241     /16-BIT ADDRESS ASSIGNED TO THE
        200     /MEMORY-MAPPED USART.
TWAIT,  MVIA    /LOAD A WITH THE FLAG MASK.
        001
        ANAM    /AND A WITH THE STATUS WORD.
        JZ      /BIT D0 OF THE WORD IS ZERO
        TWAIT   /SO WAIT FOR THE FLAG TO BE
        0       /A LOGIC ONE.
        DCXH    /THEN DECREMENT THE MEMORY I/O ADDRESS.
        POPPSW  /GET THE CHARACTER OFF OF THE STACK.
        MOVMA   /WRITE IT OUT TO THE USART.
        POPH    /POP REGISTER PAIR H OFF OF THE STACK.
        RET     /RETURN FROM THE SUBROUTINE.
```

As you have seen, the 8251 USART can be used as either an accumulator I/O device or a memory-mapped I/O device. Depending on which interfacing technique is used, either accumulator I/O or memory reference instructions have to be used to communicate with the USART.

The USART also has three control bits that may be useful to you. These are the $\overline{\text{DSR}}$ (*data set ready*) input, and the DTR (*data terminal ready*) and RTS (*request to send*) outputs. The input may be sensed by software and the outputs may be controlled with software. These may be useful if you wish to sense some sort of input (on/off) or control some device such as a teletypewriter paper-tape reader or a cassette recorder motor. For more information on the 8251 integrated circuit, refer to *The 8080 Microcomputer System User's Manual*[3].

REFERENCES

1. Rony, P. R., Larsen, D. G., Titus, J. A., and Titus, C. A. *Interfacing and Scientific Data Communications Experiments.* Howard W. Sams & Co., Inc., Indianapolis, IN, 1979.
2. Titus, C. A., Rony, P. R., Larsen, D. G., and Titus, J. A. *8080/8085 Software Design—Book 1.* Howard W. Sams & Co., Inc., Indianapolis, IN, 1978.
3. *The 8080 Microcomputer System User's Manual.* Intel Corporation, Santa Clara, CA, 1975.

2

Interrupts

In many of the programs and subroutines that have used I/O devices, the 8080 has been programmed to wait for a *flag* associated with the I/O device. This is true of the teletypewriter and crt input/output subroutines listed in Chapter 1, and many of the programs listed in the book *8080/8085 Software Design—Book 1*.[1] The reason that the 8080 is programmed to wait for the I/O devices is simply that we have assumed that the 8080 has no other important operations (tasks) to perform. For instance, in many of the programs in *8080/8085 Software Design—Book 1*[1], only a few hundred microseconds were required to "process" the characters being entered or printed. In general, this processing consisted of comparing characters, storing characters in memory, or retrieving characters from memory. However, it still takes 100 ms for a 10-character-per-second teletypewriter to send a character to or receive a character from the UART in the interface. Therefore, the 8080 spends much of its time waiting for the receiver or transmitter ready flag on the UART. This means that the 8080 is constantly *polling* the UART receiver or transmitter flag. Based on the result of this polling operation, the 8080 determines whether to input or output a character. Graf has defined polling as:

> **polling**—*1. Periodic interrogation of each of the terminals that share a communications line to determine whether it requires servicing. The multiplexer or control station (the 8080 in this case, ed.) sends a poll that has the effect of asking the selected terminal, "Do you have anything to transmit?"* 2. *A means of*

> *controlling communications lines. The communication control device will send signals to a terminal saying, "Terminal A, have you anything to send?" If not, "Terminal B, have you anything to send?" and so on. Polling is an alternative to contention. It makes sure that no terminal is kept waiting for a long time.*[2]

When a microcomputer *services* a device, it simply exchanges digital information with the device in a manner that is prescribed by some segment of the microcomputer program. The particular program segment is frequently called a *software driver*. The software driver simply transfers information between the microcomputer and the specified input/output device.

If there are a number of teletypewriters or crt's wired to the same 8080 microcomputer system, then each of the devices will have to be polled in turn. To poll these devices, the 8080 can check the state of the UART (USART) receiver and transmitter flags. If the 8080 finds a flag in the logic state that indicates that the device needs to be serviced, the 8080 stops the polling operation and begins to execute a sequence of instructions that cause the device to be serviced. Once the device has been serviced (a new character output to it or a character input from it), the 8080 polls the remaining I/O devices.

Polled operation is most useful with relatively slow devices that do not require frequent servicing, or at least can wait to be serviced. Advantage is taken of the difference in speed between the microcomputer and the polled devices, each of which might be serviced in only a fraction of a second. In 100 ms, the 8080 microcomputer can execute approximately 20,000 instructions.

As more and more I/O devices are interfaced to the 8080 microcomputer, it will take the microcomputer longer and longer to poll the devices. In fact, the polling operations can take so long that data can actually be lost. Remember, there is an *overrun flag* in the UART or USART that indicates that the received word has written over the previous word. If the previous word is not read into the microcomputer before the next word is received, this type of error will occur. Certainly, this type of situation must be avoided.

None of the previous software examples have sequentially polled two or more I/O devices. Suppose that data values have to be entered into the 8080 from two different keyboards. The ASCII characters from the keyboards must be stored in two different sections of R/W memory. A flowchart for this particular problem is shown in Fig. 2-1. An assembly language program that performs these tasks is listed in Example 2-1.

At the beginning of this program, register pair H is loaded with the R/W memory address where characters from one keyboard are

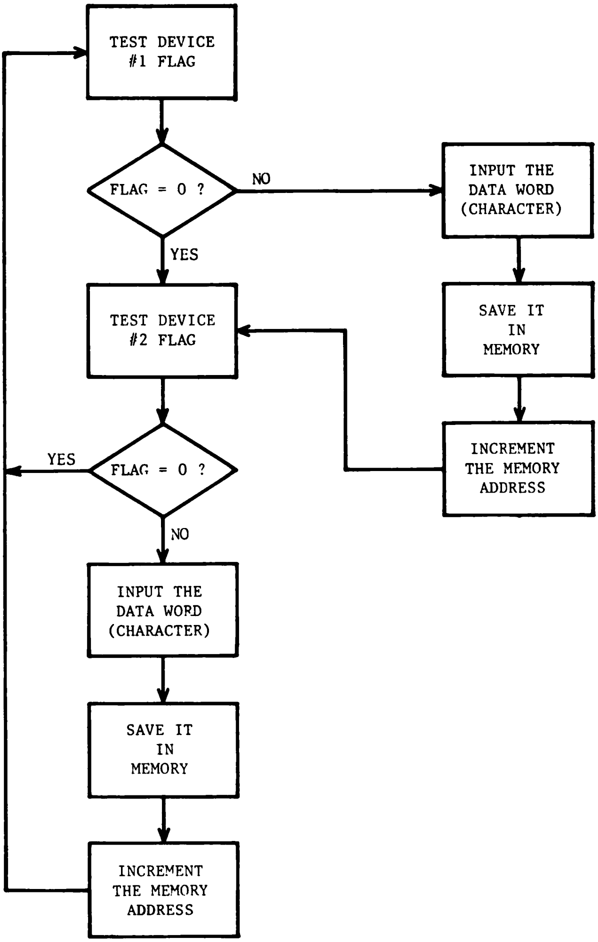

TEST DEVICE #1 FLAG
FLAG = 0 ?
NO
YES
INPUT THE DATA WORD (CHARACTER)
SAVE IT IN MEMORY
INCREMENT THE MEMORY ADDRESS
TEST DEVICE #2 FLAG
FLAG = 0 ?
YES
NO
INPUT THE DATA WORD (CHARACTER)
SAVE IT IN MEMORY
INCREMENT THE MEMORY ADDRESS

Fig. 2-1. Flowchart for polling two keyboards.

stored. Register pair D is then loaded with the R/W memory address where the characters from the other keyboard are stored. At NODEV, the 8080 checks (polls) the keyboard flag of keyboard No. 1. If the flag is a logic 1, which means that a key is pressed, the character is input from the keyboard and stored in the memory location addressed by register pair H. The memory address is then incremented and the 8080 jumps back to CHK2. This is done so that the 8080 polls the second keyboard before polling the first keyboard again.

If the 8080 determines that the flag for keyboard No. 1 is a logic 0, it will test the flag of keyboard No. 2. If this flag is also a logic 0,

Example 2-1: A Program That Polls Two Keyboards

```
/THIS PROGRAM POLLS TWO KEYBOARDS.
/WHEN A FLAG IS ONE, THE CHARACTER IS INPUT
/AND STORED IN MEMORY. TWO INDEPENDENT
/LISTS OF CHARACTERS ARE CREATED.

START,   LXIH    /LOAD A R/W MEMORY ADDRESS IN REGISTER
         BUFF1   /PAIR H WHERE THE CHARACTERS FROM
         0       /THE FIRST KEYBOARD CAN BE STORED.
         LXID    /THEN LOAD A R/W MEMORY ADDRESS IN REGISTER
         BUFF2   /PAIR D WHERE THE CHARACTERS FROM
         0       /THE SECOND KEYBOARD CAN BE STORED.
NODEV,   IN      /INPUT THE FLAG FROM THE FIRST KEYBOARD
         001     /(HEXADECIMAL 01).
         ANI     /SAVE ONLY THE KEYBOARD'S FLAG
         001     /IN A (001=HEXADECIMAL 01).
         JNZ     /THE FLAG IS A ONE, SO JUMP TO
         DEVC1   /THE SECTION OF THE PROGRAM THAT
         0       /INPUTS THE CHARACTER AND STORES IT.
CHK2,    IN      /THE FLAG FOR KEYBORD #1 IS ZERO,
         003     /SO CHECK KEYBOARD #2.
         ANI     /SAVE ONLY THE KEYBOARD'S FLAG
         001     /IN A (HEXADECIMAL 01).
         JZ      /THE FLAG IS A ZERO, SO CHECK BOTH
         NODEV   /FLAGS AGAIN.
         0
DEVC2,   IN      /THE FLAG FOR KEYBOARD #2 IS A ONE
         002     /SO INPUT THE CHAR. (002=HEXADECIMAL 02).
         STAXD   /SAVE IT IN MEMORY USING REGISTER
         INXD    /PAIR D AS THE ADDRESS.
         JMP     /THEN CHECK THE FLAG FOR THE OTHER
         NODEV   /KEYBOARD (#1) FIRST.
         0
DEVC1,   IN      /THE FLAG FOR KEYBOARD #1 IS A ONE,
         000     /SO INPUT A CHARACTER.
         MOVMA   /SAVE IT IN MEMORY USING THE ADDRESS IN
         INXH    /REGISTER PAIR H.
         JMP     /THEN CHECK THE FLAG FOR THE OTHER
         CHK2    /KEYBOARD (#2) FIRST.
         0
```

the 8080 has polled both devices and neither of them require servicing. Therefore, the 8080 executes the polling sequence of instructions again. If the flag for keyboard No. 2 is a logic 1, then it must be serviced. After the keyboard is serviced, the 8080 jumps back to NODEV, so that keyboard No. 1 is the next device to be polled.

There is nothing wrong with a microcomputer design in which I/O devices are polled. However, if there are a large number of I/O devices, it may take longer to poll the devices than actually service them. For this reason, *interrupts* are often used in microcomputer systems. If the interrupt hardware is constructed properly, and the interrupt software is written properly, it will take just as long to begin executing the *service subroutine* of one device as it will to begin executing the *service subroutine* of another device. In polled operation, it can take the microcomputer a long period of time to get to the instructions that actually cause a peripheral device to be serviced by the microcomputer.

INTERRUPT OPERATION

Graf has defined interrupt as:

> **interrupt**—*1. In a computer, a break in the normal flow of a system or routine such that the flow can be resumed from that point at a later time. The source of the interrupt may be internal or external.*[2]

Interrupt operation is a much more sophisticated mode of microcomputer operation, one that can circumvent most of the inherent disadvantages of polled operation. Thus, in polled operation:

- The microcomputer can spend much of its time sequencing through the devices wired to it.
- Once a device is serviced, it must wait its turn until all other devices are polled and, if necessary, also serviced.
- The *response time* between when a device needs servicing and when it is serviced can be substantial, at least by microcomputer standards.

In contrast, in interrupt operation:

- The microcomputer may spend much of its time processing data, displaying data, or simply executing a wait loop, waiting for a device to request servicing.
- There can exist a priority in interrupt operation. If two devices need servicing at the same time, the more important of the two devices can be serviced first.
- If a higher-priority device needs servicing while a lower-priority device is being serviced, it can interrupt the microcomputer so that it is serviced.
- The response time between when a device needs servicing and when it is serviced can be very short, even by microcomputer standards. With the 8080 microcomputer, this time is usually no more than 10 μs.
- Software becomes much more complex.

The terms *priority* and *response time* are defined in the following ways:

priority—The condition in which input/output devices are ordered in importance so that some devices take precedence over others.

response time—The time between the interrupt request by a device and the execution of the first instruction in the software driver that services it.

BASIC TYPES OF INTERRUPTS

There exist three types of interrupts: *single-line, multilevel,* and *vectored.*

single-line interrupt—An interrupt system in which there is a single interrupt line. Multiple devices must be ORed to this line. Each input to the OR gate is from an I/O device. Once it receives an interrupt, the microcomputer must poll all of the devices to determine which one generated the interrupt. This type of interrupt is often used with the Motorola 6800, the MOS Technology 6502, and the Intersil 6100.

multilevel interrupt—An interrupt system in which there exist many interrupt lines to the microcomputer, each line being tied to a separate I/O device. The microcomputer does not need to poll the devices to determine which one caused the interrupt. The 8085 has this feature.

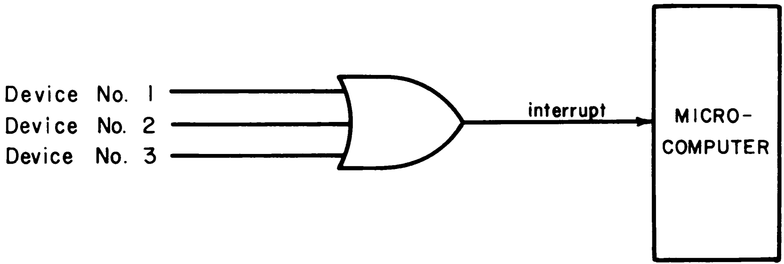
vectored interrupt—An interrupt system in which the interrupt causes a direct branch to that part of the program that services the interrupt. This is the fastest mode of interrupt operation. The 8080 and 8085 can be wired for vectored interrupts.

These three types of interrupts are shown schematically in Fig. 2-2.

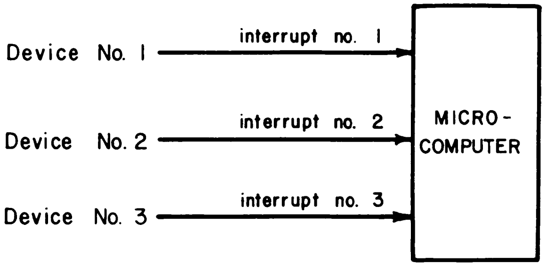
The single-line interrupt is a popular type of interrupt with microcomputers and one that is easy to implement. There exists no limit to the number of devices that can be ORed together to a single interrupt line. Three devices are shown in Fig. 2-2A. However, the more devices, the longer the time required to poll them, once an interrupt has been sensed. Even though the devices are polled, the devices can still have a priority. That is, the first device polled has the highest priority and the last device polled, once an interrupt occurs, has the lowest priority.

The multilevel interrupt, shown in Fig. 2-2B, is an effective technique if a sufficient number of interrupt pins exist on the microprocessor chip. This is rarely the case. Existing microprocessor chips have no more than five interrupt pins, as in the case of the 8085.

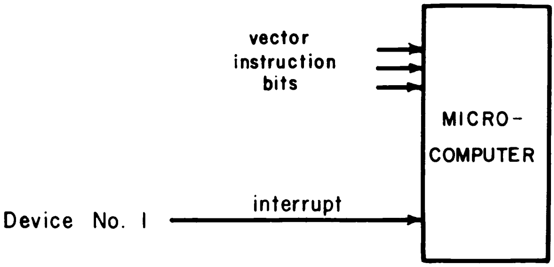
The vectored interrupt technique (Fig. 2-2C) permits direct branching to the part of the program (most likely a subroutine called an *interrupt service subroutine*) that immediately services the device that caused the interrupt. The microcomputer may output


Device No. 1
Device No. 2
Device No. 3
interrupt
MICRO-
COMPUTER

(A) Single-line interrupt.


Device No. 1
interrupt no. 1
Device No. 2
interrupt no. 2
Device No. 3
interrupt no. 3
MICRO-
COMPUTER

(B) Multilevel interrupt.


vector
instruction
bits
MICRO-
COMPUTER
Device No. 1
interrupt

(C) Vectored interrupt.

Fig. 2-2. Three different techniques for interrupting a microcomputer.

data to or input data from the I/O device when the interrupt service subroutine is executed.

For 8080 microcomputer systems, the peripheral device first generates an interrupt signal. This is really a flag output by the peripheral. The peripheral device causing the interrupt can then place a one-, two-, or three-byte instruction on the data bus, one byte at a time. The first byte, the operation code for the instruction, is written, or *jammed,* into the instruction register (IR). The remaining bytes of the instruction, if there are any, are then placed on the data bus and are interpreted by the 8080 as the data or address bytes of the instruction.

INTERRUPT INSTRUCTIONS

When an interrupt occurs, the 8080 needs to service the interrupt as quickly as possible. Therefore, a JMP or CALL instruction could be jammed into the instruction register of the 8080. If a JMP or CALL instruction op code is jammed into the instruction register, two additional bytes must be placed on the data bus by the peripheral device, the low and high address of the program (subroutine) to be jumped to (called). When all three bytes have been placed on the data bus, the 8080 actually executes the JMP or CALL instruction. One of the reasons that a CALL instruction might be used is the fact that a return address is automatically saved on the stack. This means that once the interrupting device has been serviced, the 8080 can execute a RET instruction *and return to the section of the program that was being executed when the interrupt occurred.*

Unfortunately, a considerable amount of hardware is required to jam a three-byte instruction into the 8080. Can you think of any one- or two-byte instructions that have the power of a CALL instruction? The restart instructions (RST) do. Remember, the restart instructions can be thought of as single-byte call instructions, where the address of the subroutine that is called is fixed. Table 2-1 summarizes the RST instructions and the addresses of the subroutines that are called.

One of the nice features of the op codes for the RST instructions is the fact that only three bits of the eight-bit op code vary between the eight restart instructions. All the other bits (D_7, D_6, D_2, D_1, and D_0) within the op code remain at a logic 1. This feature will be used in the design of interrupt hardware.

Only because the 8080 has been designed to accept instructions during an interrupt can the peripheral device that generated the interrupt place an instruction on the data bus and have it written into the instruction register. Although we have only discussed the

Table 2-1. A Summary of the Restart Instructions

	Op Code		Memory Address Called	
Instruction	Octal	Hex	Octal	Hex
RST0	307	C7	000 000	0000*
RST1	317	CF	000 010	0008
RST2	327	D7	000 020	0010
RST3	337	DF	000 030	0018
RST4	347	E7	000 040	0020
RST5	357	EF	000 050	0028
RST6	367	F7	000 060	0030
RST7	377	FF	000 070	0038

* Identical to hardwired RESET.

use of restart instructions by interrupting peripheral devices, a peripheral device with *simple* interrupt hardware, can place *any* single-byte instruction on the data bus when the interrupt is serviced. Depending on the function of the interrupting device, we might want to write a MOVAB, an INRE, a DCXH, an ADCM, or a NOP instruction into the instruction register when an interrupt is serviced. In another section of this chapter we will discuss the hardware required to do just this. For experiments that you can actually perform using interrupt hardware with an 8080 microcomputer, see reference 3 at the end of this chapter.

THE ENABLE AND DISABLE INTERRUPT INSTRUCTIONS

It is not uncommon for a number of I/O devices to be wired to the interrupt of an 8080-based microcomputer system. As each device interrupts the microcomputer, it is promptly serviced and then the 8080 returns to the task that it was performing when the interrupt occurred. However, if the 8080 is executing a time delay subroutine, or if it is in the process of reading information from a disk, there has to be some method of preventing I/O devices from interrupting these tasks. If the 8080 *is* interrupted while it is executing a time delay subroutine, the time delay will be somewhat lengthened by the amount of time required to service the interrupting device. If the microcomputer is interrupted while it is reading information from a disk, the disk will continue to rotate while the 8080 services the interrupt. This means that when the 8080 returns to the disk-read subroutine, it will read information from the wrong sector or track on the disk.

For this reason, there is circuitry within the 8080, and a single-byte instruction, that lets the 8080 turn a "deaf-ear" to all interrupts. The circuitry is an *interrupt flip-flop* or *flag* (the terms mean the same thing) that can be enabled or disabled by executing a single-byte microcomputer instruction.

Some definitions are in order:

interrupt flag; interrupt flip-flop—A flip-flop within the microprocessor integrated circuit that can detect an interrupt signal. It can be enabled and disabled by microcomputer software.

disable interrupt—To disable the interrupt flag within the microprocessor integrated circuit. Interrupts are ignored in this state.

enable interrupt—To enable the interrupt flag within the microprocessor integrated circuit. Interrupts are accepted and processed when the microprocessor chip is in this state.

The enable and disable interrupt instructions for the 8080 microprocessor are shown in Table 2-2.

Table 2-2: The Single-Byte Enable and Disable Interrupt Instructions

Op Code			
Octal	Hex	**Mnemonic**	**Operation**
373	FB	EI	Enable the interrupt within the microprocessor integrated circuit **following the execution of the next instruction.**
363	F3	DI	Disable the interrupt within the microprocessor integrated circuit **when this instruction is executed.**

If the interrupt is enabled, then an interrupt will be serviced immediately. However, if the interrupt is disabled by executing a DI instruction, the interrupt is ignored. Of course, if the interrupt input of the 8080 remains in the state that indicates that an interrupt needs to be serviced, then the 8080 will be interrupted as soon as the interrupt is enabled (by executing an EI instruction).

HOW THE 8080 IS ACTUALLY INTERRUPTED

For peripheral devices to interrupt the 8080, the internal 8080 interrupt flag must be enabled. This is done by executing an EI instruction. Once this is done, any device that is wired to the interrupt can interrupt the 8080. *To signal the 8080 that a device needs servicing, the interrupt pin (INT) of the 8080 must be brought to a logic 1.* However, when this occurs, the 8080 may be in the middle of executing an instruction. *Therefore, when an I/O device attempts to interrupt the 8080, the 8080 finishes executing the current instruction.* This means that the interrupting device *cannot* take the interrupt pin to a logic 1 and then simply place the op code for a RST instruction on the data bus. Instead, the interface electronics must wait until the current instruction has been executed before the RST instruction can be placed on the data bus and transferred into the instruction register. This delay may be from 0 to 9 μs if the 8080 has a 500 ns cycle time.

When an interrupt occurs, the 8080 finishes the current instruction and then generates an interrupt acknowledge ($\overline{INTA}$) signal. Of course, the $\overline{\text{INTA}}$ signal is only generated if the interrupt flag is enabled. This signal is used to coordinate the transfer of the op code onto the data bus and into the instruction register. This is the first example of *handshaking* that we have had.

handshaking—The generation of signals by two or more devices, and the recognition of the signals by the devices, so that data may be transferred between the devices in an orderly fashion.

The interrupt acknowledge ($\overline{\text{INTA}}$) *signal, which is generated by the 8080 control logic, is used to gate the one-, two-, or three-byte instruction onto the data bus and into the instruction register.* If a restart instruction is to be jammed into the instruction register, then the control logic of the 8080 generates only one $\overline{\text{INTA}}$ signal. If the op code for a three-byte instruction is to be jammed into the instruction register, then two additional $\overline{\text{INTA}}$ signals must be generated by the external control logic of the 8080 to gate the second and third bytes of the instruction into the 8080. At the same time, the interrupt signal causes the *internal interrupt flag to be disabled,* so that no other interrupts are recognized.

If a RST instruction is jammed into the 8080, it is executed and a return address is saved on the stack. The return address points to the instruction that would have been executed next if the interrupt had not occurred.

After the I/O device has been serviced, or after the interrupt has been detected, *the external I/O device flag (flip-flop) that caused the interrupt must be cleared.* This will prevent the 8080 from being interrupted by the same device many times, when the device needed to be serviced only once. After the flag has been cleared, an EI instruction can be executed so that other devices can interrupt the 8080 and be serviced. Eventually, a RET instruction is executed at the end of the interrupt service subroutine of the I/O device so that the 8080 returns to the program that was interrupted. One of the features of the 8080 microprocessor integrated circuit is that even though an EI instruction is executed, *the interrupt is not enabled until the next instruction is executed.* This means that if a RET instruction is executed after the EI instruction, the 8080 will return from the interrupt service subroutine before another device can interrupt the microcomputer.

Depending on the nature of the I/O devices that are wired to the interrupt of the microcomputer, the EI instruction may be executed at the beginning or at the end of the interrupt service subroutine. Also, the flag that caused the interrupt may be cleared at the beginning or at the end of the interrupt service subroutine.

SINGLE-LINE INTERRUPTS (POLLED INTERRUPTS)

In polled interrupts, I/O devices that need to interrupt the microcomputer are wired to a common interrupt line. When an interrupt occurs, the microcomputer has to poll each device to determine which device produced the interrupt (and needs servicing). This polling can be done very simply by reading, in one eight-bit word, the flags from up to eight I/O devices. An accumulator I/O or memory-mapped I/O instruction could be executed to do this.

By using rotate instructions or masks, it is very easy to determine which device caused the interrupt. This type of interrupt can be used on an 8080-based microcomputer, but it would have to be used in conjunction with at least one *vectored* interrupt. Polled interrupts are often used with the Motorola 6800, MOS Technology 6502, and the Intersil 6100.

VECTORED INTERRUPTS

Most microcomputers have more important things to do than monitor a single flag bit. In some cases, hundreds of flags need to be tested. In other cases, the microcomputer is performing a complex mathematical calculation that requires considerable computation time. Let us assume that a keyboard has ben interfaced to the 8080 microcomputer. The point is that most microcomputers should be interfaced in such a way that they respond only when a key is pressed on the keyboard; at all other times, the keyboard is ignored. A good typist will type five to 10 characters per second, or 100 to 200 ms per ASCII character. This is very slow by microcomputer standards, and the microcomputer can be doing many other tasks between keystrokes. However, once a key is pressed, it would be useful for the microcomputer to respond immediately. With an 8080-based microcomputer, "immediate" response can be accomplished through the use of *vectored interrupts.*

A vectored interrupt is defined as an interrupt system in which the interrupt causes a direct branch to that part of the program that services the interrupt. The necessary circuitry for an ASCII keyboard to be wired to the 8080 interrupt is shown in Fig. 2-3. Observe that an 8212 "input port" is used to jam the eight-bit op code for the RST5 instruction (octal 357, hex EF) into the instruction register. The VALID output of the keyboard is used to interrupt the 8080, and the interrupt acknowledge signal ($\overline{\text{INTA}}$) produced by the control logic of the 8080 is used to gate the eight-bit op code onto the data bus and into the instruction register. The 8212 is simply an eight-bit, three-state input port. It would have been just as easy to use devices such as the SN74126, DM8095, or SN74365 integrated circuits to gate the RST5 instruction onto the data bus.

When the 8080 chip receives an interrupt pulse, and if the interrupt flag within the chip has been previously enabled by an enable interrupt instruction, EI, *the 8080 finishes the execution of the cur-*

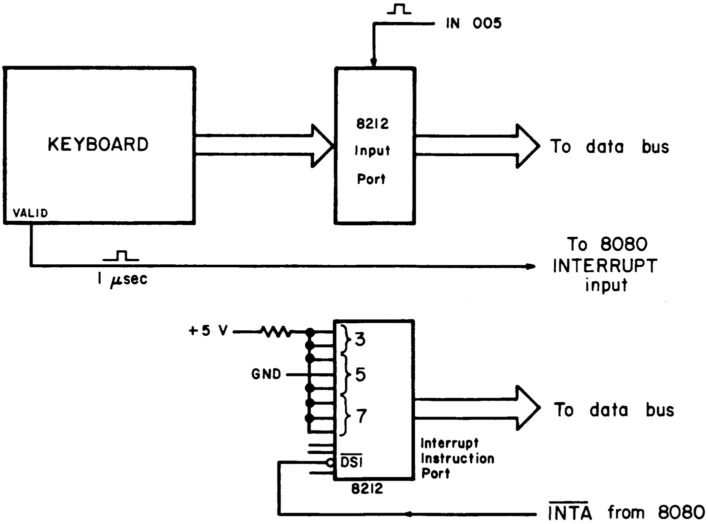


Fig. 2-3. A simple vectored interrupt for an ASCII keyboard.

rent instruction and then generates an interrupt acknowledge signal ($\overline{INTA}$) *that is used to gate the single-byte instruction, RST5* (*octal 357*)*, directly into the instruction register within the 8080 chip.* This is the only time in which you can input, using an external three-state device, directly into the instruction register rather than the accumulator or other general-purpose register. In effect, you "jam" an instruction op code into the instruction register. The hardware used to jam an instruction byte into the instruction register is shown in Fig. 2-3. It consists of an eight-bit gated driver chip that is called an *interrupt instruction register.* In Fig. 2-3, the jammed instruction is a RST5 (octal 357, hex EF), which causes the microcomputer to call the subroutine with a starting address of 000 050 (0028).

Let us examine the software required for a vectored interrupt system. Since the RST*n* instruction is a call-subroutine instruction, you must first load the stack pointer register with a read/write memory address. An enable interrupt instruction, EI, is then executed to permit the 8080 microcomputer to be interrupted by an external signal applied at the INT pin. Next, the 8080 jumps to an area of memory that we shall call MAIN TASK. This is the main program, most likely in EPROM or ROM, that will be periodically interrupted. MAIN TASK can be located anywhere in memory, but it should be located away from the interrupt service subroutine area of memory, which starts at 000 000 (0000) and continues up to approximately 000 077 (003F). Note, however, that the interrupt service subroutine for RST7 can be as long as required, because it will not "run into" any addresses that are called by any of the other restart instructions.

A program that can be used with the ASCII keyboard shown in Fig. 2-3, is listed in Example 2-2. We have assumed that the circuitry within the ASCII keyboard *debounces* the key closures.

Example 2-2: An ASCII Keyboard Program That Uses a Vectored Interrupt for Service

```
        *000 000
START,  LXISP   /LOAD THE STACK POINTER WITH A
        STACK   /R/W MEMORY ADDRESS.
        0
        EI      /ENABLE THE INTERRUPT.
        JMP     /THEN EXECUTE THE "MAIN TASK."
        MTASK
        0

        *000 050
SERVIC, IN      /A KEY IN THE KEYBOARD IS PRESSED,
        005     /SO INPUT THE ASCII CHARACTER.
        MOVBA   /SAVE THE CHARACTER IN THE B REGISTER.
        RET     /AND RETURN TO "MAIN TASK."
```

It should be clear that an interrupt from the keyboard will cause a RST5 to be jammed into the instruction register. This occurs because the 8080 generates an $\overline{\text{INTA}}$ signal. Upon execution of this instruction, the keyboard interrupt service subroutine is called, which is stored in memory starting at 000 050 (0028). This subroutine ends with a RET instruction, which permits the microcomputer to return to MAIN TASK, which was being executed when the interrupt occurred. MAIN TASK could be a simple control loop or a complicated mathematical program. Because the keyboard interrupt service subroutine is very short, the 8080 spends very little of its time actually servicing the keyboard. Note that a keyboard flag has not been polled.

The preceding program will work, but if you try to execute it, you will observe a number of operating difficulties. First, you will not be able to execute the keyboard interrupt service subroutine more than once. Why not? You have failed to reenable the interrupt flag within the 8080 chip. Remember the following rule:

> *During an interrupt machine cycle, the internal interrupt flag within the 8080 chip is first disabled, then an interrupt acknowledge signal,* $\overline{\text{INTA}}$*, is generated to permit an instruction to be jammed into the instruction register.*

The key point here is that the interrupt flag is disabled to prevent further interrupts from being detected while the microcomputer is servicing the current interrupt. *If you wish to reenable the interrupt flag, you must do so by providing an EI instruction in the inter-*

rupt service subroutine. The interrupt flag is not automatically enabled. It is also reset or disabled by an external RESET pulse applied to the 8080.

It is common practice to provide an enable interrupt instruction (EI) immediately before the RET instruction at the end of the interrupt service subroutine. Since the *interrupt flag does not become enabled until after the next instruction has been executed,* the 8080 can return to MAIN TASK before another interrupt is accepted by the 8080 chip. If this capability were not provided, there would be the danger that the 8080 would fill read/write memory with return addresses because interrupts would interrupt interrupt service subroutines before the 8080 had the chance to return to MAIN TASK.

In all other respects, you treat interrupt service subroutines as normal subroutines. If the contents of the registers are important, you use PUSH and POP instructions to save and restore the registers. A typical interrupt service subroutine would appear as shown in Fig. 2-4. Note that the EI instruction is located just before the RET instruction.

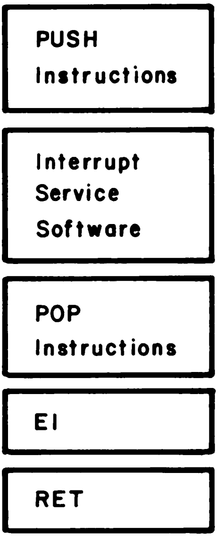


Fig. 2-4. Block diagram for a typical interrupt service subroutine.

Since there exist only eight memory locations between the vector address 000 050 (0028) and the next vector address 000 060 (0030), you will not be able to fit in four PUSH instructions, four POP instructions, one EI instruction, one RET instruction, and the keyboard service instructions without encroaching on the next one or two vector subroutine locations. Instead, you must place a jump instruction, starting at 000 050 (0028), that transfers control to an area of memory where you will have more room for software. At the end of the interrupt service subroutine, the RET instruction will still return program control to the point where MAIN TASK was interrupted. The relationship between the MAIN TASK, the vector jump instruction at 000 050 (0028), and the keyboard interrupt service subroutine is shown in Fig. 2-5. The interface for the keyboard could have been more complex. For example, priority interrupts could have been used. In the preceding example, there was little incentive to do so.

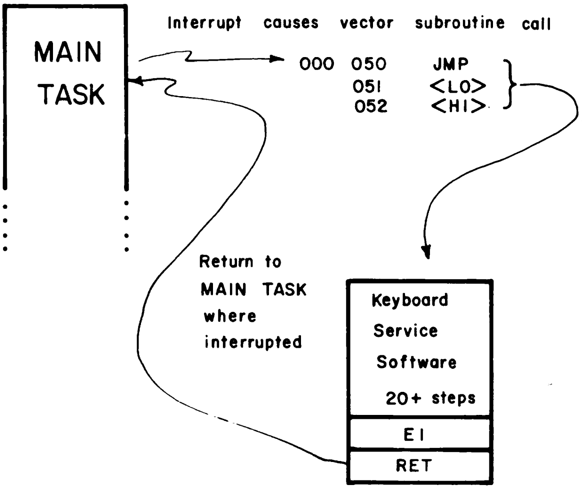

MAIN TASK
Interrupt causes vector subroutine call
000 050 JMP
051 <LO>
052 <HI>
Return to MAIN TASK where interrupted
Keyboard Service Software
20+ steps
EI
RET

Fig. 2-5. Jumping to the interrupt service subroutine.

VECTORED AND POLLED INTERRUPTS

Many 8080 users use vectored interrupts, where a different RST instruction is dedicated to each peripheral device wired to the interrupt. However, there is a limit to the number of interrupt devices that can use RST instructions, since the 8080 can only execute eight RST instructions. Also, an eight-bit, three-state driver must be used by each device to gate the appropriate RST instruction onto the data bus. For these reasons, some users use polled interrupts, simply because they require less hardware. An interface for three polled interrupt devices is shown in Fig. 2-6. We have assumed that the ASCII keyboard is no longer wired to the 8080 interrupt. We have also assumed that the "interrupt instruction port" is still wired to the 8080 $\overline{\text{INTA}}$ signal so that when an interrupt does occur, a RST5 is jammed into the instruction register. Refer to Fig. 2-3 for details.

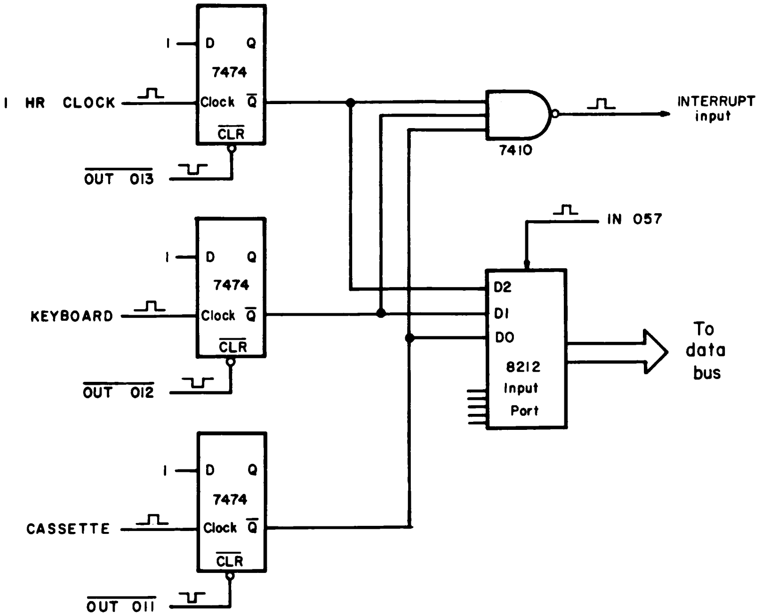

1
D Q
7474
1 HR CLOCK
Clock Q̄
CLR
OUT 013
1
D Q
7474
KEYBOARD
Clock Q̄
CLR
OUT 012
1
D Q
7474
CASSETTE
Clock Q̄
CLR
OUT 011
7410
INTERRUPT input
IN 057
D2
D1
D0
8212 Input Port
To data bus

Fig. 2-6. An interface for three polled interrupt devices that uses one RST instruction.

Three peripheral devices use the interrupt: a high-speed cassette, a medium-speed keyboard, and a very-low-speed clock. It will be assumed that the cassette flag indicates that an eight-bit word can be input from the cassette. The keyboard flag indicates when a key is pressed, and the one-hour clock flag indicates that one hour has elapsed since the clock last produced a clock pulse.

The software that *polls* these devices is listed in Example 2-3. Only the interrupt service subroutine for the cassette is included in the listing.

At START, the stack pointer is loaded with a R/W memory address. This is done because a RST5 will be jammed into the instruction register when an interrupt is acknowledged by the 8080. When this instruction (RST5) is executed, a return address is saved on the stack, so a stack area must be available. The three interrupt flags are then cleared by the three output instructions. After the flags are cleared, the interrupt is enabled (EI) and then the MAIN TASK is executed.

The interrupt service subroutine is stored in memory starting at 000 050 (0028). The interrupt service subroutine is stored here because the interface electronics jam a RST5 into the instruction register when an interrupt is acknowledged. When an interrupt does occur, program control is vectored to 000 050 (0028). The flag bits

Example 2-3: The Interrupt Service Subroutine That Polls Three Devices

```
        *000 000
START,  LXISP   /LOAD THE STACK POINTER WITH AN ADDRESS
        RWMEM   /FOR R/W MEMORY.
        0
        OUT     /CLEAR THE CASSETTE'S INTERRUPT
        011     /FLAG (HEXADECIMAL 09).
        OUT     /CLEAR THE KEYBOARD'S INTERRUPT
        012     /FLAG (HEXADECIMAL 0A).
        OUT     /AND CLEAR THE ONE-HOUR CLOCK'S
        013     /INTERRUPT FLAG (HEXADECIMAL 0B).
        EI      /ENABLE THE INTERRUPT.
        •       /AND EXECUTE THE "MAIN TASK."
        •
        •

        *000 050
SERVIC, IN      /INPUT THE FLAGS FROM THE DEVICES
        057     /WIRED TO THE INTERRUPT (HEXADECIMAL 2F).
        MOVBA   /SAVE THE FLAGS IN B.
        ANI     /SAVE ONLY THE CASSETTE'S FLAG
        001     /(HEXADECIMAL 01).
        JZ      /THE FLAG IS A ZERO, SO READ THE
        CASRD   /DATA FROM THE CASSETTE.
        0
        MOVAB   /GET THE FLAGS BACK.
        ANI     /SAVE ONLY THE KEYBOARD'S FLAG
        002     /(HEXADECIMAL 02).
        JZ      /THE FLAG IS A ZERO, SO INPUT
        KEYIN   /THE ASCII CHARACTER.
        0
        MOVAB   /GET THE FLAGS BACK.
        ANI     /SAVE ONLY THE ONE-HOUR CLOCK'S FLAG
        004     /(HEXADECIMAL 04).
        JZ      /THE FLAG IS A ZERO, SO TURN
        CLOCK   /A LIGHT ON AND TURN A HEATER OFF.
        0
        HLT     /WHAT CAUSED THE INTERRUPT?

CASRD,  OUT     /CLEAR THE CASSETTE'S FLAG, WHICH
        011     /CAUSED THE INTERRUPT (HEXADECIMAL 0A).
        IN      /INPUT THE DATA FROM THE CASSETTE
        103     /(HEXADECIMAL 43).
        MOVMA   /SAVE IT IN MEMORY.
        INXH    /INCREMENT THE MEMORY ADDRESS.
        EI      /ENABLE THE INTERRUPT.
        RET     /AND THEN RETURN.
```

are then input and stored in register B so that they can be polled. Using this type of interface electronics, the 8080 can only determine which device generated the interrupt by polling or examining the flags.

After the flags are saved in the B register, the first ANI instruction tests only the cassette flag contained in the A register. If the cassette flag is 0, the JZ to CASSRD (cassette read) is executed. If the flag is a logic 1, the cassette has not caused the interrupt, so the flags are moved from the B register to the A register and then the next ANI instruction tests only the keyboard flag. If the keyboard flag is 0, the JZ to KEYIN is executed, so that the ASCII character transmitted by the keyboard is input. If the keyboard flag is a logic 1, then the flag for the one-hour clock is polled. If the clock flag is a logic 1, the 8080 executes the HLT instruction. If the clock flag is a logic 0, the JZ to CLOCK is executed.

Why has the 8080 been programmed to execute a HLT at the end of the polling software? This is executed because some device interrupted the 8080, but when the 8080 polled the devices to determine which needed servicing, none of the flags were in the proper state. This indicates that something is wrong in the interface hardware. Unfortunately, this problem can be caused by a number of different situations, such as improper or inadequate power supply filtering, poor power supply decoupling, noise on the microcomputer inputs, improper grounding, etc. These kinds of problems can be very difficult to locate. Instead of programming the microcomputer to halt, it could be programmed so that "UNIDENTIFIED INTERRUPT" is printed on a crt or teletypewriter. Refer to Chapter 3 of *8080/8085 Software Design–Book 1*[1] (see Example 3-19 for a start).

If the cassette flag is a logic 0 in this example, the JZ to CASRD is executed. At CASRD, the output instruction clears the cassette interrupt flag. The data word from the cassette is then input into register A and stored in memory. The memory address is then incremented. The EI instruction "enables the interrupt," but it is really not enabled until after the RET instruction has been executed. When the RET instruction is executed, the interrupt is enabled and the 8080 returns to the section of MAIN TASK that was being executed when the interrupt occurred.

When an instruction is jammed onto the data bus, the 8080 acknowledges the interrupt. You might ask, however, why the content of the memory location addressed by the 16-bit address on the address bus does not interfere with the instruction being jammed into the instruction register by the interface electronics. When the 8080 acknowledges the interrupt, neither the $\overline{\text{MEMR}}$ or $\overline{\text{MEMW}}$ line to memory is pulsed. Therefore, no information from memory is placed on the data bus. When the interrupt is acknowledged, the previous instruction has been completely executed. Therefore, when the $\overline{\text{INTA}}$ signal is produced, the 8080 thinks that it is fetching an instruction from memory, but the instruction really comes from the interface electronics of the interrupting device–$\overline{\text{INTA}}$ is pulsed rather than $\overline{\text{MEMR}}$!

Do you see any problems with the interrupt service subroutine as it is now written (Example 2-3)? The problems have to do with the content of the registers when an interrupt occurs. What is in the B register or register pair H when an interrupt occurs? There is no way of knowing, since an interrupt can occur at any time. To remedy this, register pairs B and H should be saved on the stack, depending on what registers are used by the KEYIN and CLOCK subroutines. The interrupt service subroutine listed in Example 2-4 saves these register pairs on the stack.

Example 2-4: An Improved Three-Device Polled Interrupt Subroutine

```
        *000 050
SERVIC, PUSHPSW /SAVE A AND THE FLAGS ON THE STACK.
        PUSHB   /THEN SAVE REGISTER PAIR B ON THE STACK.
        IN      /INPUT THE FLAGS FROM THE DEVICES
        057     /WIRED TO THE INTERRUPT (HEXADECIMAL 2F).
        MOVBA   /SAVE THE FLAGS IN B.
        ANI     /SAVE ONLY THE CASSETTE'S FLAG
        001     /(HEXADECIMAL 01).
        JZ      /THE FLAG IS A ZERO, SO READ THE
        CASRD   /DATA FROM THE CASSETTE.
        0
        MOVAB   /GET THE FLAGS BACK.
        ANI     /SAVE ONLY THE KEYBOARD'S FLAG
        002     /(HEXADECIMAL 02).
        JZ      /THE FLAG IS A ZERO, SO INPUT
        KEYIN   /THE ASCII CHARACTER.
        0
        MOVAB   /GET THE FLAGS BACK.
        ANI     /SAVE ONLY THE ONE-HOUR CLOCK'S FLAG
        004     /(HEXADECIMAL 04).
        JZ      /THE FLAG IS A ZERO, SO TURN
        CLOCK   /A LIGHT ON AND TURN A HEATER OFF.
        0
        HLT     /WHAT CAUSED THE INTERRUPT?

CASRD,  PUSHH   /PUSH REGISTER PAIR H ONTO THE STACK.
        LHLD    /LOAD REGISTER PAIR H WITH AN ADDRESS
        POINT   /STORED IN TWO CONSECUTIVE MEMORY
        0       /LOCATIONS.
        OUT     /THEN CLEAR THE CASSETTE'S FLAG, WHICH
        011     /CAUSED THE INTERRUPT (HEXADECIMAL 0A).
        IN      /INPUT THE DATA FROM THE CASSETTE
        103     /(HEXADECIMAL 43).
        MOVMA   /SAVE IT IN MEMORY.
        INXH    /INCREMENT THE MEMORY ADDRESS.
        SHLD    /THEN SAVE THE NEW ADDRESS IN TWO CON-
        POINT   /SECUTIVE MEMORY LOCATIONS.
        0
        POPH    /POP REGISTER PAIR H OFF OF THE STACK.
        POPB    /POP REGISTER PAIR B OFF OF THE STACK.
        POPPSW  /POP A AND THE FLAGS OFF OF THE STACK.
        EI      /ENABLE THE INTERRUPT.
        RET     /AND THEN RETURN.
```

When an interrupt occurs, the 8080 is vectored to this new interrupt service subroutine, starting at 000 050 (0028). The 8080 immediately saves the PSW and register pair B on the stack. The individual I/O device flags are then polled. Assuming that the cassette requires service, the cassette flag will be a logic 0. Therefore, the 8080 jumps to CASRD. At CASRD, registered pair H is pushed onto the stack. Register pair H is then loaded with the 16-bit address that was previously stored in the two R/W memory locations assigned the symbolic address POINT. The interrupt flag is then cleared, the data word is read from the cassette, and then stored in the memory location addressed by register pair H. The address in register pair H is incremented and stored back in R/W memory. Register pairs H and B and the PSW are then popped off of the stack and the interrupt is reenabled. The RET instruction then returns program control to the section of MAIN TASK that was interrupted.

The interrupt service subroutine now appears to be completely *transparent* to the program that was interrupted, because none of the values in any of the registers from MAIN TASK are disturbed. When the interrupt service subroutine is executed, the required values are saved on the stack and are then popped off of the stack before control is returned to MAIN TASK. As always, it is very important that the "proper care and feeding of the stack" be maintained. Note that the KEYIN and CLOCK subroutines must also end with the POPB, POPPSW, EI, and RET sequence of instructions for proper operation.

Has a priority been set up in the polling software? What device has the highest priority in the interrupt service subroutine? The cassette has the highest priority, simply because its flag is tested first. The clock has the lowest priority and the keyboard has a priority between these two devices. If a floppy disk is added to the microcomputer interrupt, what should its priority be? Since the floppy disk has a higher data transmission/reception rate than the cassette, it should have the highest priority. Assuming that one of the floppy disk flags is assigned to bit D_3 of the polled input port, the interrupt service subroutine listed in Example 2-5 could be used to poll all four interrupt devices.

PRIORITY INTERRUPTS

In the previous interrupt software examples, the priority of each device was determined by the order in which it was polled. A higher-priority device was polled before a lower-priority device.

priority interrupts—Priority interrupts are interrupts that are ordered in importance so that some interrupting devices take

Example 2-5: Adding a Higher Priority Device to the Interrupt Service Subroutine

```
        *000 050
SERVIC, PUSHPSW /SAVE A AND THE FLAGS ON THE STACK.
        PUSHB   /PUSH REGISTER PAIR B ON THE STACK.
        IN      /INPUT THE FLAGS FROM THE DEVICES
        057     /WIRED TO THE INTERRUPT (HEXADECIMAL 2F).
        MOVBA   /SAVE THE FLAGS IN B.
        ANI     /SAVE ONLY THE FLOPPY DISK'S FLAG
        010     /(HEXADECIMAL 08).
        JZ      /THE FLAG IS A ZERO, SO READ
        FLOPPY  /THE DATA FROM THE DISK.
        0
        MOVAB   /GET THE FLAGS BACK.
        ANI     /SAVE ONLY THE CASSETTE'S FLAG
        001     /(HEXADECIMAL 01).
        JZ      /THE FLAG IS A ZERO, SO READ THE
        CASRD   /DATA FROM THE CASSETTE.
        0
        MOVAB   /GET THE FLAGS BACK.
        ANI     /SAVE ONLY THE KEYBOARD'S FLAG
        002     /(HEXADECIMAL 02).
        JZ      /THE FLAG IS A ZERO, SO INPUT
        KEYIN   /THE ASCII CHARACTER.
        0
        MOVAB   /GET THE FLAGS BACK.
        ANI     /SAVE ONLY THE ONE-HOUR CLOCK'S FLAG
        004     /(HEXADECIMAL 04).
        JZ      /THE FLAG IS A ZERO, SO TURN
        CLOCK   /A LIGHT ON AND TURN A HEATER OFF.
        0
        HLT     /WHAT CAUSED THE INTERRUPT?
```

precedence over others. They are used whenever a number of interrupts can occur at the same time, or whenever there is a need to determine which interrupting devices are the most important.

As you have already seen, changing the priority of the software polled I/O devices is very simple; the order in which the devices are polled by the software simply has to be changed.

HARDWARE PRIORITY INTERRUPTS

Interrupt priorities may also be generated using hardware. Hardware priority interrupts are very important when a number of interrupting devices are connected to the microcomputer and all require relatively fast service. The "gimmick" employed is simple. *Each interrupting device generates its own restart instruction, RSTn, which, when written into the instruction register of the 8080, causes an immediate vector to memory location 000 000, 000 010, 000 020, 000 030, 000 040, 000 050, 000 060, or 000 070 (in hexadecimal,* *0000, 0008, 0010, 0018, 0020, 0028, 0030, or 0038*). In addition, priority is automatically assigned by the hardware, so that device 7 has higher priority than device 6, which has a higher priority than device 5, etc. In other words, if > represents priority, then:

$$7>6>5>4>3>2>1>0$$

The circuit that you would use for hardware priority, vectored interrupts is shown in Fig. 2-7.

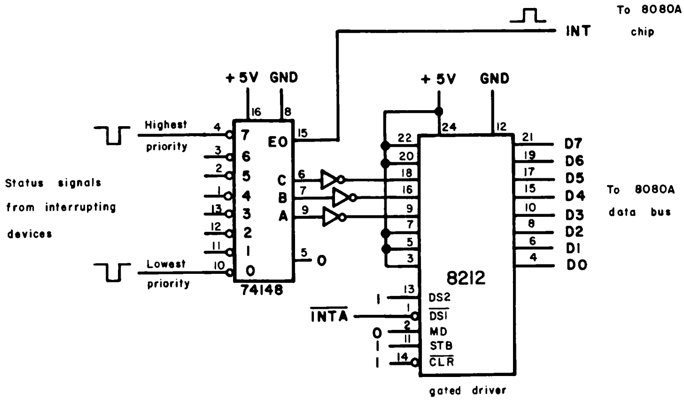

To 8080A chip
INT
+5V
GND
16
8
Highest priority
Status signals from interrupting devices
Lowest priority
4
3
2
1
13
12
11
10
7
6
5
4
3
2
1
0
EO
C
B
A
15
6
7
9
5
0
74148
+5V
GND
24
12
22
20
18
16
9
7
5
3
21
19
17
15
10
8
6
4
D7
D6
D5
D4
D3
D2
D1
D0
To 8080A data bus
8212
13
DS2
DS1
MD
STB
CLR
INTA
gated driver

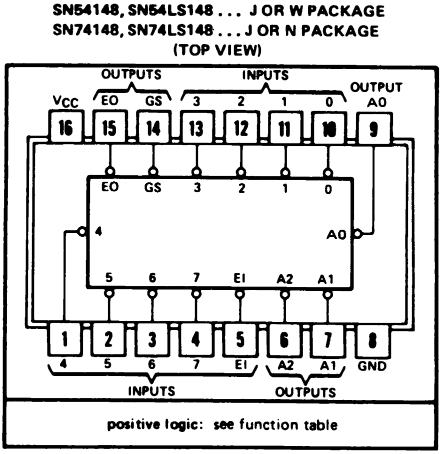
Fig. 2-7. An eight-level, hardware priority, vectored interrupt circuit that generates eight different restart instructions.

The 74148 eight-line-to-three-line priority encoder integrated circuit is a 16-pin chip that has the pin configuration and truth table shown in Fig. 2-8. Note that the data inputs and data outputs are active at the low logic level. The 74148 chip will accept up to eight logic 0 inputs from flags, such as those shown in Fig. 2-7, and will output *the binary code for the highest numbered input that is at logic 0.* For example, if you have simultaneous interrupt requests from both device 5 and device 7, device 7 has the higher priority, and the 74148 chip and inverters will supply an octal 7 for the middle octal digit in the restart instruction, 3*n*7. This vectors the 8080 to memory location 000 070 (0038). The RST0 instruction is not often used since its only effect is to reset the microcomputer and start MAIN TASK again.

Fig. 2-7 has been simplified for clarity. The necessary flags and flag-clearing lines are not shown. Additional hardware refinements could be added to the circuit to make it more sophisticated. This would include an additional 7442 decoder to generate the flag-clearing pulses without using OUT instructions, and a mask register so that various devices could be masked *off* or *on* (disabled and enabled) via external hardware. The modified circuit is shown in

Fig. 2-9. It is a very sophisticated priority interrupt interface that provides great flexibility in the use of vectored interrupts in conjunction with an 8080-based microcomputer.

To use the interrupt controller shown in Fig. 2-9, you must first decide which devices will be allowed to interrupt the microcomputer and which will not. You develop an eight-bit mask pattern in which interrupting devices are assigned a logic 1 and noninterrupting devices are assigned a logic 0. These eight bits are then output from the 8080 to the two 7475 latches shown in the left of Fig. 2-9. An OUT 030 instruction is used for this purpose. Bit position D_7 corresponds to interrupting device 7, which has the highest priority and can generate a vector to address 000 070 (0038). Those devices that are masked will probably use a status register input port to request service, as shown previously in this chapter (see the polling sections).


SN54148, SN54LS148 . . . J OR W PACKAGE
SN74148, SN74LS148 . . . J OR N PACKAGE
(TOP VIEW)
OUTPUTS
INPUTS
OUTPUT
VCC
EO
GS
3
2
1
0
AO
16
15
14
13
12
11
10
9
EO
GS
3
2
1
0
4
AO
5
6
7
EI
A2
A1
1
2
3
4
5
6
7
8
4
5
6
7
EI
A2
A1
GND
INPUTS
OUTPUTS
positive logic: see function table

'148, 'LS148
FUNCTION TABLE

INPUTS									OUTPUTS				
EI	0	1	2	3	4	5	6	7	A2	A1	A0	GS	EO
H	X	X	X	X	X	X	X	X	H	H	H	H	H
L	H	H	H	H	H	H	H	H	H	H	H	H	L
L	X	X	X	X	X	X	X	L	L	L	L	L	H
L	X	X	X	X	X	X	L	H	L	L	H	L	H
L	X	X	X	X	X	L	H	H	L	H	L	L	H
L	X	X	X	X	L	H	H	H	L	H	H	L	H
L	X	X	X	L	H	H	H	H	H	L	L	L	H
L	X	X	L	H	H	H	H	H	H	L	H	L	H
L	X	L	H	H	H	H	H	H	H	H	L	L	H
L	L	H	H	H	H	H	H	H	H	H	H	L	H

Fig. 2-8. The pin configuration and truth table for the SN74148 priority encoder integrated circuit.

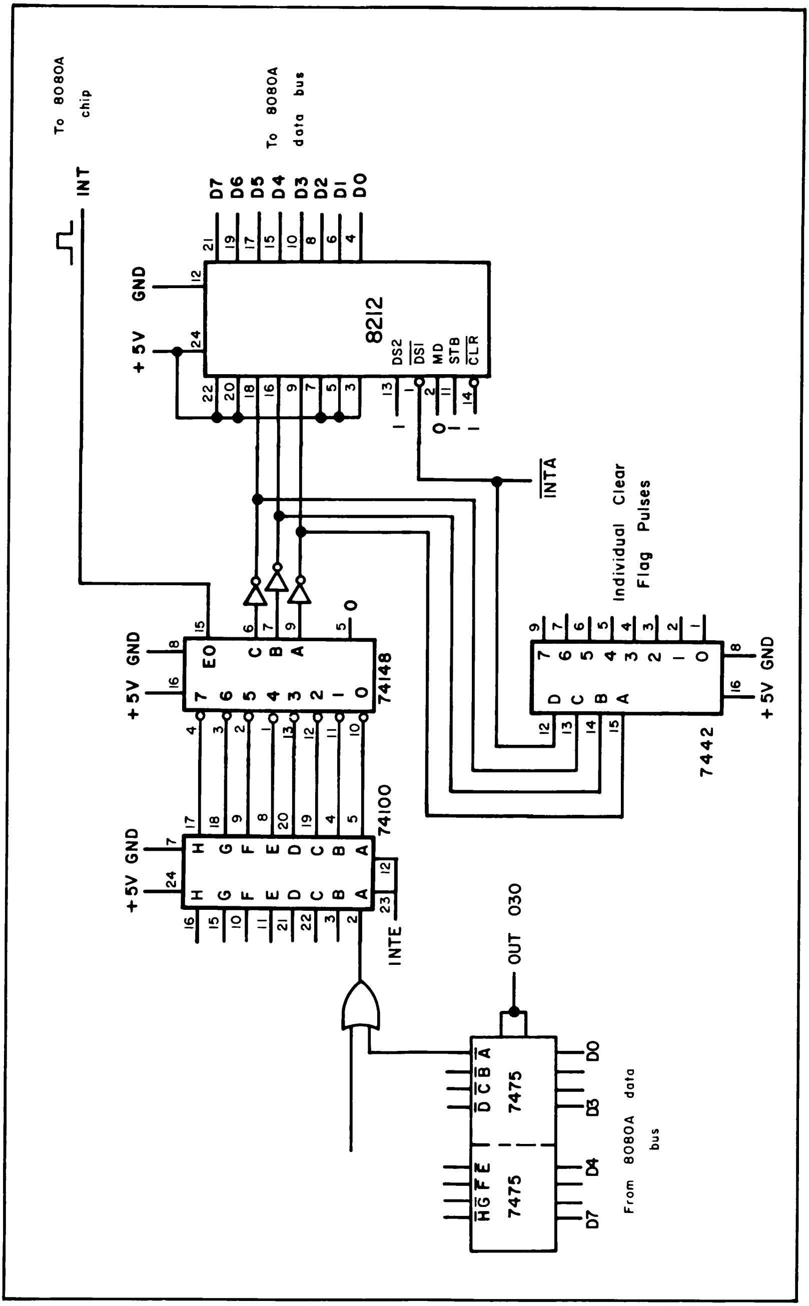

To 8080A chip
INT
To 8080A data bus
D7 D6 D5 D4 D3 D2 D1 D0
+5V GND
8212
DS2 DSI MD STB CLR
INTA
Individual Clear Flag Pulses
74148
EO C B A
7 6 5 4 3 2 1 0
7442
D C B A
+5V GND
74100
H G F E D C B A
INTE
OUT 030
7475
From 8080A data bus
D0 D3 D4 D7

Fig. 2-9. A sophisticated eight-level, hardware priority, vectored interrupt controller.

Interrupt requests from the flags are gated with the OR gates (one is shown in the left of Fig. 2-9), and the outputs are wired to the 74100 eight-bit latch. Whenever the interrupt flag within the 8080 chip is enabled, INTE (interrupt enable; a signal output by the 8080 chip) is high and enables the 74100 chip. The actions of the 74148 priority encoder and interrupt instruction port have been previously described.

When the interrupt is received by the 8080 chip, it disables the interrupt, and the INTE output goes to a logic 0, latching any interrupts present at the 74100 chip. The $\overline{\text{INTA}}$ signal not only inputs the RST*n* instruction op code, *it also pulses the 7442 decoder so that a pulse is produced that clears the flip-flop associated with the vectored interrupt currently being serviced.*

The program listed in Example 2-6 assumes that the same one-hour clock, keyboard, and cassette are still interfaced to the 8080 microcomputer. Each device has now been assigned a different

Example 2-6: Interrupt Service Subroutines for Three Vectored Devices

```
        *000 000
START,  LXISP   /LOAD THE STACK POINTER WITH AN ADDRESS
        RWMEM   /FOR R/W MEMORY.
        0
        OUT     /CLEAR THE CASSETTE'S INTERRUPT
        011     /FLAG (HEXADECIMAL 09).
        OUT     /CLEAR THE KEYBOARD'S INTERRUPT
        012     /FLAG (HEXADECIMAL 0A).
        OUT     /AND CLEAR THE ONE-HOUR CLOCK'S
        013     /INTERRUPT FLAG (HEXADECIMAL 0B).
        EI      /ENABLE THE INTERRUPT AND
        •       /EXECUTE THE REMAINDER OF THE
        •       /"MAIN TASK."

        *000 050
CLOCK,  OUT     /CLEAR THE ONE-HOUR CLOCK'S
        013     /INTERRUPT FLAG (HEXADECIMAL 0B).
        EI      /THEN ENABLE THE INTERRUPT FOR OTHER DEVICES.
        PUSHPSW /SAVE A AND THE FLAGS ON THE STACK.
        JMP     /THEN EXECUTE THE REMAINDER
        CLK1    /OF THE CLOCK INTERRUPT SERVICE
        0       /SUBROUTINE.
        *000 060
KEYIN,  PUSHPSW /SAVE A AND THE FLAGS ON THE STACK.
        IN      /INPUT THE EIGHT-BIT ASCII
        000     /KEY CODE (HEXADECIMAL 00).
        ANI     /THEN REMOVE ANY PARITY
        177     /BITS FROM THE CODE (HEXADECIMAL 7F).
        JMP     /THEN EXECUTE THE REMAINDER OF
        KEYIN1  /KEYIN INTERRUPT SERVICE
        0       /SUBROUTINE.
        *000 070
CASRD,  PUSHPSW /SAVE A AND THE FLAGS ON THE STACK.
        PUSHH   /THEN SAVE REGISTER PAIR H ON THE STACK.
```

```
          LHLD      /LOAD REGISTER PAIR H WITH AN ADDRESS
          POINT     /STORED IN TWO CONSECUTIVE MEMORY
          0         /LOCATIONS.
          OUT       /THEN CLEAR THE CASSETTE'S FLAG, WHICH
          011       /CAUSED THE INTERRUPT (HEXADECIMAL 0A).
          IN        /INPUT THE DATA FROM THE CASSETTE
          103       /(HEXADECIMAL 43).
          MOVMA     /SAVE IT IN MEMORY.
          INXH      /INCREMENT THE MEMORY ADDRESS.
          SHLD      /THEN SAVE THE NEW ADDRESS IN TWO CON-
          POINT     /SECUTIVE MEMORY LOCATIONS.
          0
          POPH      /AND POP REGISTER PAIR H OFF OF THE STACK.
          POPPSW    /POP A AND THE FLAGS OFF OF THE STACK.
          EI        /ENABLE THE INTERRUPT.
          RET       /AND THEN RETURN.

CLK1,     IN        /INPUT THE DATA FROM THE EIGHT
          034       /SENSE SWITCHES (HEXADECIMAL 1C).
          ANI       /THEN SAVE ONLY THE STATUS OF THE THREE
          142       /SWITCHES (D6, D5, AND D1; HEXADECIMAL 62).
          CPI       /ARE ALL THE SWITCHES IN THE LOGIC
          142       /ONE STATE?
          JNZ       /NO, THEN LEAVE THE HEATER ON
          LVEON     /SO THAT THE TANK CONTINUES TO
          0         /WARM UP.
          OUT       /THE TANK IS WARM ENOUGH, SO TURN
          101       /THE LIGHT ON
          OUT       /AND THE HEATERS OFF
          102       /(HEXADECIMAL 41 AND 42).
LVEON,    POPPSW    /GET A AND THE FLAGS OFF OF THE STACK.
          RET       /AND THEN RETURN.

KEYIN1,   OUT       /CLEAR THE KEYBOARD'S INTERRUPT FLAG
          012       /(HEXADECIMAL 0A).
          CPI       /SEE IF THE KEY CODE IS FOR THE
          104       /D KEY, OR THE DECREASE COMMAND.
          JZ        /IT WAS A D FOR DECREASE, SO
          DEC       /DECREMENT THE ADDRESS IN H BY 1.
          0
          CPI       /SEE IF THE KEY CODE IS FOR THE
          111       /I KEY, OR THE INCREASE COMMAND.
          JNZ       /IT WAS NOT THE I KEY, IGNORE THE
          NOTINC    /COMMAND
          0
          INRH

NOTINC,   POPPSW    /POP A AND THE FLAGS OFF OF THE STACK.
          EI        /ENABLE THE INTERRUPT.
          RET       /AND THEN RETURN.

DEC,      DCRH      /DECREMENT THE ADDRESS IN H.
          POPPSW    /POP THE PSW OFF OF THE STACK.
          EI        /ENABLE THE INTERRUPT.
          RET       /AND THEN RETURN.
```

priority (and restart instruction). The one-hour clock jams a RST5 into the microcomputer, the keyboard jams a RST6, and the cassette jams a RST7 into the 8080. The interrupt service subroutines for all three devices are listed in Example 2-6.

At the beginning of this program, the stack pointer is loaded with a R/W memory address. The three interrupt flags are then cleared by the three OUT instructions and then the interrupt is enabled. The 8080 then executes the remaining instructions in MAIN TASK. When an interrupt occurs, each interrupting device will jam its own restart instruction onto the data bus, so that the 8080 vectors to the appropriate interrupt service subroutine.

The interrupt service subroutine for the one-hour clock starts at 000 050 (0028). What restart instruction must the one-hour clock jam into the microcomputer? The one-hour clock must be wired to the interrupt controller or interface so that a RST5 is jammed into the microcomputer. The keyboard has a higher priority than the one-hour clock, so it jams a RST6 into the microcomputer. Because of this, its interrupt service subroutine starts at 000 060 (0030). The cassette has the highest priority, so it causes the 8080 to be vectored to 000 070 (0038) when it needs servicing.

If the one-hour clock interrupts the 8080, the microcomputer is vectored to 000 050 (0028). Starting at this address, the 8080 clears the one-hour clock interrupt flag and then reenables the interrupt. This means that the keyboard or cassette can interrupt the servicing of the clock, if required. After the interrupt is reenabled, the PSW is saved on the stack and the 8080 jumps to the remainder of the clock interrupt service subroutine at CLK1. The entire interrupt service subroutine cannot be stored in memory between 000 050 and 000 060 (0020 to 0028), so some instructions are stored between these two addresses, and the remainder are stored in another section of memory. Note that a JMP was not stored at 000 050 (0028), so control was not immediately transferred to the interrupt service subroutine.

When the 8080 jumps to CLK1, the remainder of the subroutine is executed. The task that we assigned to the one-hour clock was arbitrarily chosen. At CLK1, the status of eight switches is input into the A register. The state of the switches wired to bits D_6, D_5, and D_1 is then saved in register A. If these three switches are all in the logic 1 state, the heater for some tank of liquid has been left on for a sufficient period of time. If one, two, or all three of the switches are in the logic 0 state, then the heater should remain on.

How long will it be before the 8080 checks the state of the switches again? It will be one hour, since the clock generates an interrupt once an hour.

It does not matter, for this discussion, what the temperature of the tank is, or even what is in the tank. After these two control functions are performed, register A and the flags are popped off of the stack, and then the 8080 returns to the MAIN TASK.

Since the one-hour clock has the lowest priority, the clock flag is cleared as soon as the 8080 vectors to 000 050 (0028). After the flag is cleared, the interrupt is reenabled. *It is extremely important to realize that the interrupt must only be reenabled after the clock flag is cleared.*

The KEYIN subroutine is the interrupt service subroutine for the ASCII keyboard that is interfaced to the 8080. When a key is pressed, the 8080 is interrupted and is vectored to 000 060 (0030). The 8080 then saves the A register and the flags on the stack, and then inputs the eight-bit ASCII character from the keyboard into the A register. The ANI instruction sets the parity bit (D_7) to 0. The JMP instruction transfers control to the remainder of the subroutine. Again, rather than just save a JMP at 000 060 (0030), several memory locations are better utilized by actually executing a few of the interrupt service instructions. If a JMP was simply stored at 000 060 (0030), there would be five unused memory locations between the high address byte of the JMP instruction and the interrupt service subroutine for the device with the next higher priority. Therefore, it is more efficient to actually execute some of the instructions here.

When the 8080 jumps to KEYIN1, the keyboard interrupt flag is cleared and the content of the A register is compared to the ASCII values for the D and I keys. If the D key on the keyboard has been pressed, then the content of the H register is decremented by 1. If the I key is pressed, the content of the H register is incremented by 1. Again, this is an arbitrary task that we wanted the 8080 to perform when either the D or I key on the keyboard is pressed. If neither of these keys is pressed, the 8080 pops the A register and the flags off of the stack, reenables the interrupt, and returns to MAIN TASK. Note that the cassette cannot interrupt the servicing of the keyboard, since the EI instruction is executed at the end of the interrupt service subroutine.

The cassette interrupt service subroutine is slightly different from the other two. The cassette has been assigned the RST7 instruction, so there are no peripherals that have a higher priority. The interrupt service subroutine can therefore be stored in memory starting at 000 070 (0038). The entire interrupt service subroutine can be stored in memory, starting at this address, because no devices can vector the 8080 to a higher address. The CASRD subroutine is very similar to the polled interrupt service subroutine that was discussed in another section of this chapter. The only difference is that the PUSHB and POPB instructions have been eliminated from this version of the interrupt service subroutine.

If the interrupt hardware in Fig. 2-9 is used to interface the one-hour clock, the keyboard, and the cassette to the 8080 interrupt, what changes would have to be made to Example 2-6? At the beginning of the MAIN TASK, some instructions would have to be executed so that all of the interrupt flags are cleared. This must be done because the state of the flip-flops cannot be determined when power is first applied to the microcomputer system. Therefore, a number of OUT instructions are executed, each clearing an individual flop-flop. No OUT instructions have to be executed in the interrupt service subroutines because the interrupt hardware (the 7442) generates pulses that can be used to clear the flags. This means that 5 μs are saved in each interrupt service subroutine.

THE 8085 AND INTERRUPTS

When Intel designed the 8085, they added a number of fine interrupt features. There are five interrupt inputs on the 8085 integrated circuit, one of which is INTR. This input has the same function as the INT input on the 8080 microprocessor integrated circuit. If the 8085 interrupt is enabled (by executing an EI instruction), the 8085 will be interrupted by taking the INTR pin to a logic 1. The 8085 will acknowledge the interrupt by generating a logic 0 pulse on the $\overline{\text{INTA}}$ pin. This signal can be used to gate instructions onto the data bus and into the instruction register (IR).

The 8085 also has four vectored priority interrupts built right into the microprocessor integrated circuit. The addresses for the interrupt service subroutines that the 8085 calls when one of these pins is taken to a logic 1, are listed in Table 2-3.

Table 2-3. The Vector Addresses for the 8085 Vectored Priority Interrupts

	Vector Address	
Interrupt	Octal	Hex
RST7.5	000 074	003C
RST6.5	000 064	0034
RST5.5	000 054	002C
TRAP	000 044	0024

The priority of the five 8085 interrupt inputs is as follows: TRAP> RST7.5>RST6.5>RST5.5>INTR. For the moment, we will limit our discussion to the three interrupts, RST7.5, RST6.5, and RST5.5.

When one of these pins on the 8085 microprocessor integrated circuit is taken to a logic 1, a restart instruction is *automatically* written into the instruction register and executed. No external priority encoders or interrupt instruction registers are required. The subroutine stored at the appropriate vector address will then be executed so that the interrupt is serviced. However, unlike the usual RST*n* operations, there are no RST7.5, RST6.5, or RST5.5 instructions that we can also execute or use in a program. They are only executed when one of the three interrupt pins on the integrated circuit is taken to a logic 1. As you would expect, however, the 8085 must execute an EI instruction before it can be interrupted by a logic 1 applied to either the RST7.5, RST6.5, RST5.5, or INTR pins. However, before the 8085 can be interrupted by RST7.5, RST6.5, or RST5.5, we must also enable the individual interrupts with an *interrupt mask.*

The interrupt mask can be used to enable or disable any combination of the three RST*n*.5 interrupts. The interrupt mask register is loaded with the three least-significant bits of the A register when a SIM (*S*et *I*nterrupt *M*ask) instruction is executed. The SIM instruction is special to the 8085, and it does not appear in the 8080 instruction set. Bit D_0 of the A register represents the mask for RST5.5, bit D_1 for RST6.5, and bit D_2 for RST7.5. The content of bits D_0, D_1, and D_2 will only be transferred into the interrupt mask register if bit D_3 of the A register is a logic 1 when the SIM instruction is executed. This is a *mask enable* control bit. As you know from Chapter 1, the SIM instruction is also used to output information to the SOD pin of the 8085.

To enable one of the interrupts, the mask must contain a logic 0; to disable an interrupt, the mask must contain a logic 1.

As can be seen from Table 2-4, bit D_3 of the A register must be a logic 1 when the SIM instruction is executed if it is desired to change the content of the interrupt mask register.

Let us use the previous example (Example 2-6) to demonstrate how the interrupt mask register is set. The only changes that have

Table 2-4. The Interrupt Mask Bits for the 8085

D_3	D_2	D_1	D_0	Interrupts Enabled/Disabled
1	0	0	0	RST7.5, 6.5, and 5.5 enabled.
1	0	0	1	RST7.5 and 6.5 enabled, 5.5 disabled.
1	0	1	0	RST7.5 and 5.5 enabled, 6.5 disabled.
1	0	1	1	RST7.5 enabled, 6.5 and 5.5 disabled.
1	1	0	0	RST6.5 and 5.5 enabled, 7.5 disabled.
1	1	0	1	RST6.5 enabled, 7.5 and 5.5 disabled.
1	1	1	0	RST5.5 enabled, 7.5 and 6.5 disabled.
1	1	1	1	RST7.5, 6.5, and 5.5 disabled.
0	X	X	X	Interrupt mask is not changed.
X = May be a logic 1 or a logic 0.				

been made to Example 2-6 are due to the new interrupt capabilities of the 8085. The new interrupt service subroutines are listed in Example 2-7.

Example 2-7: Programming the Interrupt Mask Register of the 8085

```
        *000 000
START,  LXISP    /LOAD THE STACK POINTER WITH AN ADDRESS
        RWMEM    /FOR R/W MEMORY.
        0
        OUT      /CLEAR THE CASSETTE'S INTERRUPT
        011      /FLAG (HEXADECIMAL 09).
        OUT      /CLEAR THE KEYBOARD'S INTERRUPT
        012      /FLAG (HEXADECIMAL 0A).
        OUT      /AND THEN CLEAR THE ONE-HOUR CLOCK'S
        013      /INTERRUPT FLAG (HEXADECIMAL 0B).
        MVIA     /ENABLE THE THREE INTERRUPTS BY SETTING
        010      /THE MASK BITS TO 0 (XXXX1000).
        SIM      /TRANSFER REGISTER A TO THE INTERRUPT MASK.
        EI       /ENABLE THE INTERRUPT AND
        •        /THEN EXECUTE THE REMAINDER OF THE
        •        /"MAIN TASK."

        *000 054
CLOCK,  JMP      /JUMP UP TO THE INTERRUPT SERVICE
        CLK1     /SUBROUTINE FOR THE ONE-HOUR CLOCK.
        0

        *000 064
KEYIN,  JMP      /JUMP UP TO THE INTERRUPT SERVICE
        KEYIN1   /SUBROUTINE FOR THE ASCII KEYBOARD.
        0

        *000 074
CASRD,  PUSHPSW  /SAVE A AND THE FLAGS ON THE STACK.
        PUSHH    /THEN SAVE REGISTER PAIR H ON THE STACK.
         •       /GET A VALUE FROM THE CASSETTE AND
         •       /SAVE IT IN MEMORY.
         •
        POPH     /POP REGISTER PAIR H OFF OF THE STACK.
        OUT      /CLEAR THE FLAG THAT CAUSED THE
        011      /CASSETTE TO INTERRUPT THE 8085.
DONE,   MVIA     /NOW ENABLE ALL OF THE INTERRUPTS
        010      /BY MEANS OF THE INTERRUPT MASK.
        SIM
        POPPSW   /POP A AND THE FLAGS OFF OF THE STACK.
        EI       /ENABLE THE INTERRUPT.
        RET      /AND THEN RETURN.

CLK1,   PUSHPSW  /SAVE A AND THE FLAGS.
        MVIA     /THEN ENABLE JUST RST7.5 AND
        011      /RST6.5 (XXXX1001).
        SIM      /SET THE INTERRUPT MASK.
        OUT      /CLEAR THE FLAG THAT CAUSED
        013      /THE CLOCK TO INTERRUPT THE 8085.
```

```
          EI       /AND ENABLE THE INTERRUPT.
          •        /NOW SERVICE THE SWITCHES,
          •        /HEATERS, AND LIGHT.
          POPPSW   /RESTORE A AND THE FLAGS.
          RET      /RETURN TO "MAIN TASK."

KEYIN1,   PUSHPSW  /SAVE A AND THE FLAGS.
          IN       /INPUT THE EIGHT-BIT ASCII
          000      /CHARACTER.
          •        /THEN COMPARE THE CHARACTER TO
          •        /ASCII I AND/OR ASCII D AND TAKE
          •        /THE APPROPRIATE ACTIONS.
          OUT      /CLEAR THE FLAG THAT CAUSED
          012      /THE KEYBOARD TO INTERRUPT THE 8085.
          JMP      /THEN LOAD THE INTERRUPT MASK.
          DONE     /POP A AND THE FLAGS, REENABLE THE
          0        /INTERRUPT, AND RETURN TO "MAIN TASK."
```

In Example 2-7, we have not included many of the actual device service instructions in the interrupt service subroutines. However, additional instructions have been added to the interrupt service subroutines so that the 8085 interrupt mask register is programmed with the proper values.

At the start of the program, the stack pointer is loaded with a R/W memory address, and the three interrupt flags are cleared by the three OUT instructions. The A register is then loaded with 00001000, which will cause the RST7.5, RST6.5, and RST5.5 interrupts to be enabled when the SIM instruction is executed. However, no devices can interrupt the microcomputer until the EI instruction is executed. Once this is done, the 8080 executes the remainder of MAIN TASK. When an interrupt does occur, the 8080 will be vectored to either 000 054, 000 064, or 000 074 (002C, 0034, or 003C).

If the one-hour clock interrupts the microcomputer, the 8085 is vectored to 000 054 (002C) and the JMP to CLK1 is executed. At CLK1, the A register and the flags are saved on the stack and then the A register is loaded with 00001001. The SIM instruction writes this value into the interrupt mask register because bit D_3 of the word is a logic 1. The result is that interrupt RST5.5 is masked out. This prevents the one-hour clock from interrupting the execution of its own interrupt service subroutine. This is not very likely, since the clock only interrupts the microcomputer every hour. However, in some instances, this can be a troublesome problem. More will be said about this problem in the next section of this chapter.

Regardless of which interrupt is enabled or disabled, the 8085 still cannot be interrupted by the keyboard or cassette at this time, since the internal interrupt flag has not been reenabled. After the SIM instruction is executed, the 8085 clears the clock interrupt flag by executing the OUT 013 instruction. When the EI instruction is executed, either the keyboard or the cassette can interrupt the execution of the clock interrupt service subroutine. Once the interrupt is reenabled, the 8085 performs the control functions that were previously explained (monitor switches, turn a heater and a light *on* or *off*). After these operations have been performed, the 8085 returns to MAIN TASK.

The cassette and keyboard interrupt service subroutines are slightly different from the clock interrupt service subroutine. In the cassette and keyboard subroutines, the interrupt is reenabled *after* the device has been serviced. As you already know, this means that no device can interrupt the 8085 while it is servicing either of these devices. When the 8085 has performed the appropriate service operations, the A register is loaded with 00001000, so that all of the RST*n*.5 interrupts are enabled by the interrupt mask. The SIM instruction transfers the content of the A register to the interrupt mask register, and then the internal interrupt flag is reenabled when the EI instruction is executed. The 8085 then returns to MAIN TASK.

The 8085 can also execute another instruction that is not contained in the 8080 instruction set. This is the RIM (*R*ead *I*nterrupt *M*ask) instruction. By executing this instruction, the 8085 can read the state of the three interrupt masks into the A register along with the logic states of the three RST*n*.5 inputs to the 8085 integrated circuit. This instruction *does not* have to be used in many applications.

The last interrupt that we will discuss is TRAP. If the TRAP pin of the 8085 is taken to a logic 1, the 8085 is vectored to address 000 044 (0024). *The TRAP interrupt cannot be disabled by executing a DI instruction or by changing the interrupt mask register with a SIM instruction.* There is no way to disable the TRAP interrupt. Also, remember that TRAP has the highest priority of *all* 8085 interrupts. Finally, there is no TRAP instruction that the 8085 can execute, so that the subroutine at 000 044 (0024) is executed. This is the same situation as with the interrupts, RST7.5, RST6.5, and RST5.5. For more information on the 8085 interrupt structure, refer to *MCS-85 User's Manual.*[4]

PRIORITY INTERRUPT SOFTWARE TIMING

As the final topic in this chapter, let us consider the timing required to execute priority interrupt software. Assume that we have only two interrupting devices: high-priority device 7 and low-priority device 2. Each device generates its own restart instruction byte, which causes a vector to either address 000 070 (0038) or 000 020 (0010), respectively. Also assume that the high-priority device interrupts the main program, MAIN TASK, on a regular basis and is quickly serviced by the software. Device 2, the low-priority device, is assumed to interrupt on an irregular schedule. It requires considerable time to service. For example, device 2 could be another microcomputer that is transferring blocks of data into our microcomputer.

The most important software is the MAIN TASK software that is executed whenever the external devices are not being serviced. If the software were not important, it would not be the "main task" performed by the microcomputer. Early in MAIN TASK, we locate the stack pointer with an LXISP instruction and enable the interrupt by executing an EI instruction.

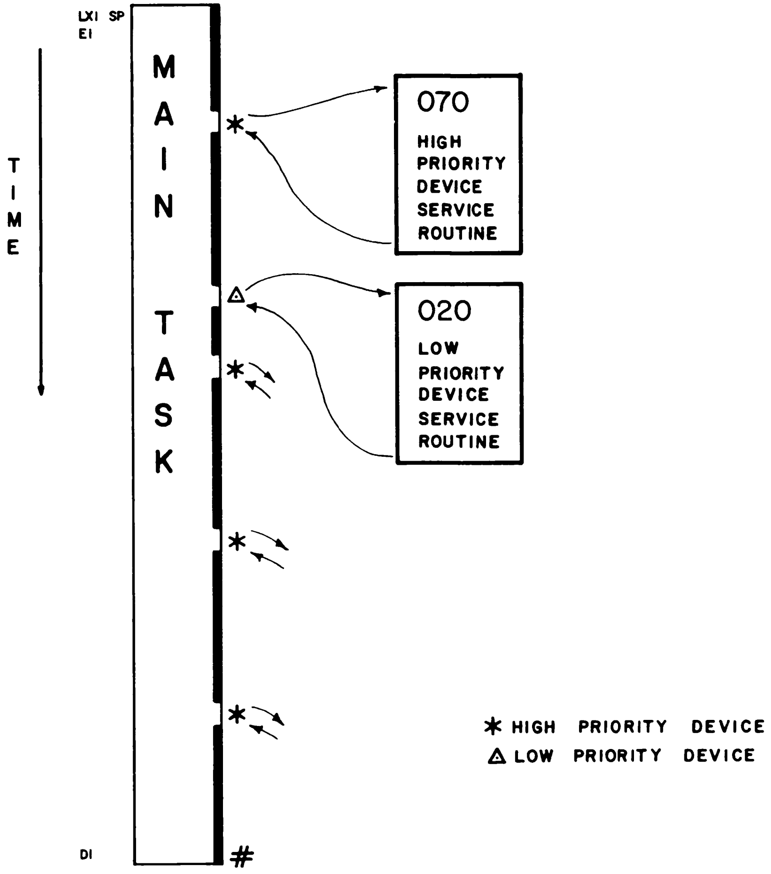

LXI SP
EI
TIME
MAIN TASK
070
HIGH PRIORITY DEVICE SERVICE ROUTINE
020
LOW PRIORITY DEVICE SERVICE ROUTINE
DI
#
* HIGH PRIORITY DEVICE
△ LOW PRIORITY DEVICE

Fig. 2-10. Program execution time line for MAIN TASK.

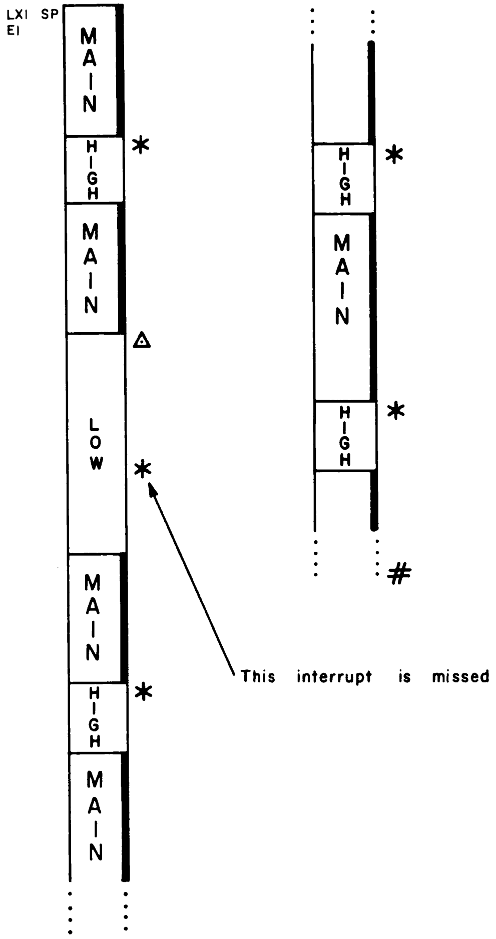


Fig. 2-11. Program execution time line for MAIN TASK.

Since interrupts can occur at any time, both PUSH and POP instructions are required in the interrupt service subroutines. Such instructions will save and restore any registers that are altered in the subroutines. The execution of the software can be graphically represented by a *time line*, as shown in Fig. 2-10.

Notice that the high-priority device has interrupted MAIN TASK four times, whereas the low-priority device has interrupted the microcomputer only once. The high-priority device interrupts on a regular basis, as shown by its spacing on the MAIN TASK time line. The heavy line indicates when the interrupt is enabled.

Fig. 2-11 shows a more realistic time line. The time line in Fig. 2-10 is somewhat deceptive since only the time spent in MAIN

TASK is shown. It is more correct to show the real time spent in both MAIN TASK and the subroutines. In Fig. 2-11, the MAIN TASK starts to operate and is interrupted by the high-priority device. After executing the high-priority device interrupt service subroutine, control is returned to MAIN TASK, which is interrupted by the low-priority device later on the time line. Control is eventually returned to MAIN TASK, which is then interrupted at repeated intervals by the high-priority device. Clearly, it takes considerably longer to reach the end of MAIN TASK when it is repeatedly interrupted by peripheral devices. During a critical timing period, such repeated interruptions would be disastrous if we are relying on programmed time-delay loops to generate time delays.

We have assumed that the high-priority device interrupts on a regular basis. It probably tried to interrupt the execution of the low-priority interrupt service subroutine shown in Fig. 2-11. If high had a higher priority than low, why didn't an interrupt occur? The answer is that the interrupt flag within the 8080 chip *was not enabled during the execution of the low-priority interrupt service subroutine.* In our first attempt at writing the interrupt service software, we forgot to take this possibility into account. As a result, data or signals from the high-priority device were lost during the execution of the low-priority interrupt service subroutine. We can correct our software easily by placing the enable interrupt instruction, EI, at the beginning of the low-priority interrupt service subroutine. We can also design hardware to store data or signals that occur during a missed interrupt.

By moving the enable interrupt instruction, EI, to the beginning of the low-priority interrupt service subroutine, we may encounter a new problem: the execution of the low-priority interrupt service subroutine may be chopped up, as shown in Fig. 2-12. To emphasize the point, we have assumed that the high-priority device interrupts the execution of the low-priority interrupt service subroutine twice, chopping the low-priority software into three pieces. With the low-priority device software so split up, we must inquire whether we are able to complete executing the low-priority software *before the low-priority device generates another interrupt.* It is entirely possible for the low-priority device to interrupt the microcomputer while it is still trying to service the last interrupt request from the low-priority device. While the interrupt response is fast, the actual execution time may be somewhat slower than the time required for a single pass through the interrupt service subroutine. This is a consequence of the fact that we can interrupt the servicing of interrupts. It is very easy for a microcomputer to become *interrupt bound;* i.e., it spends all of its time checking and

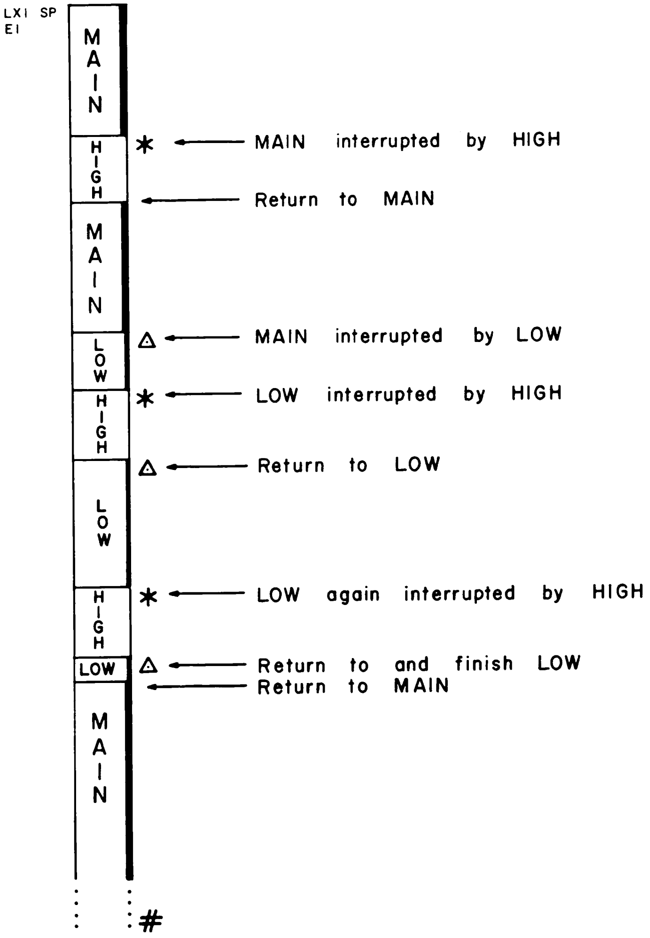

LXI SP
EI
MAIN
HIGH
MAIN
LOW
HIGH
LOW
HIGH
LOW
MAIN
* ← MAIN interrupted by HIGH
← Return to MAIN
△ ← MAIN interrupted by LOW
* ← LOW interrupted by HIGH
△ ← Return to LOW
* ← LOW again interrupted by HIGH
△ ← Return to and finish LOW
← Return to MAIN
#

Fig. 2-12. Program execution time line for an interrupt that interrupts an interrupt.

servicing interrupts and has no time left for its MAIN TASK software.

In our MAIN TASK software, we may wish to prevent interrupts from occurring during sensitive time-delay software or complex time-dependent tasks or calculations. The disable interrupt instruction, DI, allows the microcomputer to be immune to external interrupts. Such a situation is shown in Fig. 2-13. The interrupt flag is disabled to permit a critical task to be performed, and then is reenabled. Unfortunately, the time line for the execution of MAIN TASK shows that an interrupt from the high-priority device was missed. Without additional, and usually complex, hardware as a backup, it is very easy to lose signals or data from interrupting devices while the interrupt flag is disabled. The important point here is that *we do not know when an external device will interrupt MAIN TASK, and we cannot be sure that it will not try to do so during the period of time that the interrupt is disabled.* How do we circumvent this problem? It is not easy, and this is why we must use a great deal of caution when using interrupts.

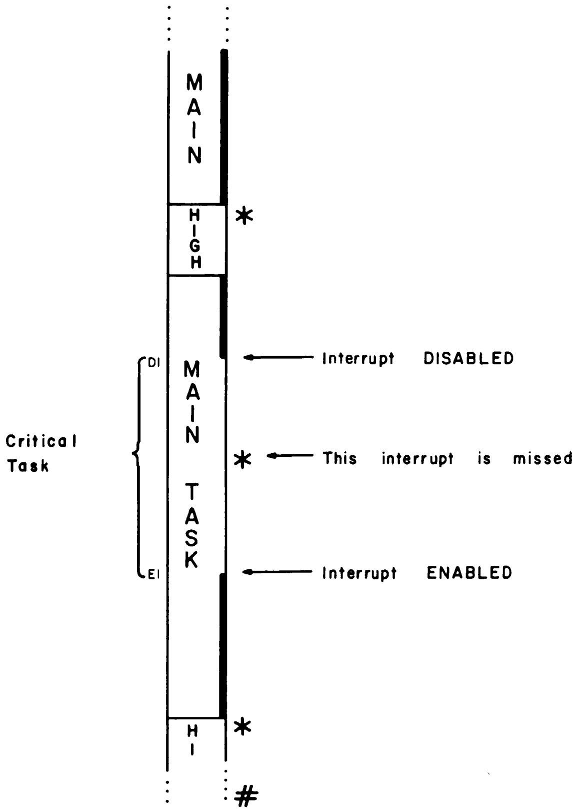

M A I N
H I G H
*
DI
M A I N T A S K
Critical Task
Interrupt DISABLED
*
This interrupt is missed
EI
Interrupt ENABLED
H I
*
#

Fig. 2-13. Using DI and EI instructions during time-critical tasks.

Another type of interrupt which may be of interest is a *time-oriented interrupt*. Only one interrupt is used: a clock. The clock interrupts the microcomputer every 10 ms, or other reasonable time period. When interrupted, the microcomputer uses a look-up table to determine which devices to check to see if they need service. Some devices are always checked, while other slower devices might be checked once every 1000 times. This is a useful technique, but it requires considerable amounts of software to work well.

The newer support integrated circuits for 8080-type microcomputers permit multibyte instructions to be written into the instruction register during an interrupt. This means that a complete three-byte jump or call instruction can be inserted into the microprocessor integrated circuit. The ability to "jam" a three-byte instruction eliminates the need to use the restart instructions, and associated vector locations, and provides you with much greater flexibility in the use of hardware and software. To jam a three-byte instruction into the 8080 means that the support logic for the 8080 integrated circuit must generate *three* $\overline{\text{INTA}}$ pulses (one for each byte of the instruction). The logic to do this is complex and, therefore, infrequently done. If the 8228 system controller is used with the 8080, it is a little easier to do. The Intel 8259 programmable interrupt controller allows you to perform direct calls to interrupt service subroutines, but it is a complex device, not for the beginner.

We shall finish this chapter with some final notes of caution. Interrupts are difficult to debug. Since interrupts can occur at almost any time, typical software debugging programs are difficult to apply; most are ineffective. Special diagnostic software is required to test interrupts in specific applications. *If you can avoid the use of interrupts, do so.* Spend your valuable time on other non-interrupt approaches, if possible. Your efforts will be well rewarded.

REFERENCES

1. Titus, C. A., Rony, P. R., Larsen, D. G., and Titus, J. A. *8080/8085 Software Design—Book 1.* Howard W. Sams & Co., Inc., Indianapolis, IN, 1978.
2. Graf, R. F. *Modern Dictionary of Electronics.* Howard W. Sams & Co., Inc., Indianapolis, IN, 1977.
3. Larsen, D. G., Rony, P. R., and Titus, J. A. *Introductory Experiments in Digital Electronics and 8080A Microcomputer Programming and Interfacing —Book 2.* Howard W. Sams & Co., Inc., Indianapolis, IN, 1978.
4. *MCS-85 User's Manual.* Intel Corporation, Santa Clara, CA, 1978.
5. Ulrickson, R. W. "Real-Time Systems Often Use Interrupts." *Electronic Design.* May 10, 1977, pp 80-84.
6. Baldridge, R. L. "Interrupts Add Power, Complexity to μC-System Design." *EDN,* August 5, 1977, pp 67-73.
7. Shima, M. and Blacksher, R. "Improved Microprocessor Interrupt." *Electronic Design,* April 26, 1978, pp 96-100.
8. Vittera, J. "Treat Interrupts With Care in Single-Chip μC Systems." *EDN,* October 20, 1977, pp 59-61.
9. Vittera, J. "Handling Multilevel Subroutines and Interrupts in Microcomputers." *Computer Design,* January 1978, pp 109-115.

3

Interrupt Applications

Now that we have discussed both how the 8080 can be interrupted and the signals and instructions required to service these interrupts, we will discuss the application of interrupts toward solving interfacing problems. As you will see, the solutions involve both hardware and software.

A REAL-TIME CLOCK

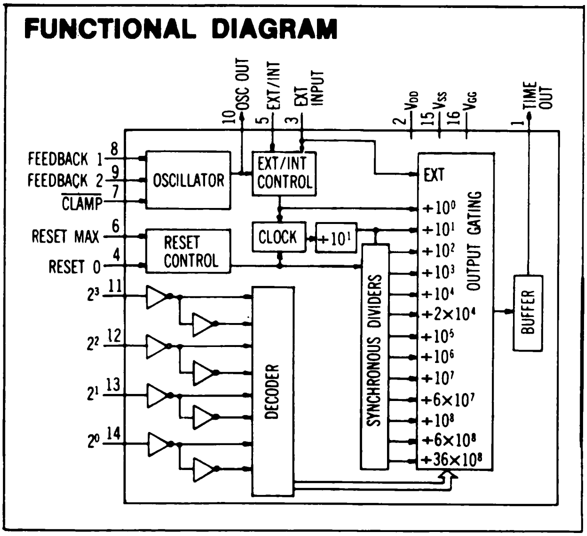
In some applications, the 8080 may be required to output data to a peripheral device at certain predetermined intervals; for example, once every 23 milliseconds or once every 13.2 seconds. As we have already seen, it is possible to program the 8080 with a series of instructions that produce time delays of such duration. After the delay occurs, a data word could be output to the peripheral device and then another delay period could be commenced. However, for most applications this is impractical because the microcomputer has many other tasks to perform, and it will be difficult to calculate the time required to perform these various tasks. For this reason, we will use a *real-time clock* that is wired to the 8080 interrupt. By using a real-time clock, an interrupt can be generated every 23 milliseconds, 13.2 seconds, or other interval, which means that time delay subroutines are no longer required in our software, and we do not have to calculate the time required to perform other tasks.

What is a real-time clock? A real-time clock is a peripheral device that keeps time, *regardless of what the microcomputer is doing*. In fact, a real-time clock can continue to keep time, even when the microcomputer halts, simply because it is a peripheral device that operates independently from the microcomputer. When the real-time clock *times-out,* which means that the real-time clock has timed a predetermined interval, the 8080 is interrupted. At this time, the 8080 can service appropriate peripheral devices. Depending on the hardware design, the 8080 may also have to service the real-time clock.

The "heart" of our real-time clock will be the MOSTEK Corporation MK5009 MOS counter time-base circuit (Fig. 3-1). This integrated circuit has four programming inputs that determine the clock frequency of its output (TIME OUT, pin 1). When a 1-MHz quartz crystal is wired to the MK5009, the frequencies listed in Table 3-1 can be obtained by programming the MK5009 with four-bit binary words from 0000 through 1000 (on pins 14 through 11). For additional information on the MK5009, refer to Appendix A.

Of course, from what we know about the minimum time required to service an interrupt (at a minimum, save all of the registers on the stack), it would be impossible for the 8080 to service a peripheral device that interrupts the microcomputer every one or 10 microseconds.

The microcomputer can service a peripheral device that interrupts it every 100 microseconds, but only if the interrupt service subroutine is kept as short as possible. The highest frequency


FUNCTIONAL DIAGRAM
10 OSC OUT
5 EXT/INT
3 EXT INPUT
2 V_{DD}
15 V_{SS}
16 V_{GG}
1 TIME OUT
FEEDBACK 1 8
FEEDBACK 2 9
$\overline{\text{CLAMP}}$ 7
RESET MAX 6
RESET 0 4
2^3 11
2^2 12
2^1 13
2^0 14
OSCILLATOR
EXT/INT CONTROL
CLOCK
$\div 10^1$
RESET CONTROL
DECODER
SYNCHRONOUS DIVIDERS
OUTPUT GATING
EXT
$\div 10^0$
$\div 10^1$
$\div 10^2$
$\div 10^3$
$\div 10^4$
$\div 2\times10^4$
$\div 10^5$
$\div 10^6$
$\div 10^7$
$\div 6\times10^7$
$\div 10^8$
$\div 6\times10^8$
$\div 36\times10^8$
BUFFER

Courtesy MOSTEK Corp.

Fig. 3-1. Functional diagram for the MOSTEK MK5009 MOS counter time-base circuit.

(shortest period) that you can use to interrupt your microcomputer will really depend on the complexity of the operations to be performed when an interrupt occurs.

Using the MK5009, can we generate time delays of 23 milliseconds or 13.2 seconds? Yes, but the solution requires both hardware and software.

The Hardware/Software Real-Time Clock Solution

The hardware that we will use for the real-time clock interface is shown in Fig. 3-2. As you can see from this diagram, a four-bit output port (OUT 305) is used to store an appropriate programming value for the MK5009. The TIME OUT output of the MK5009 is used to clock a D flip-flop, and the output of the flip-flop ($\overline{Q}$) drives one of the priority-encoder-based interrupt interfaces shown in the previous chapter. An OUT 306 will cause this flip-flop to be cleared once the interrupt is serviced. Since this hardware will not interrupt the 8080 every 15 milliseconds or 20 seconds (see Table 3-1), the interrupt service subroutine must somehow determine whether or not a delay of the required duration has occurred.

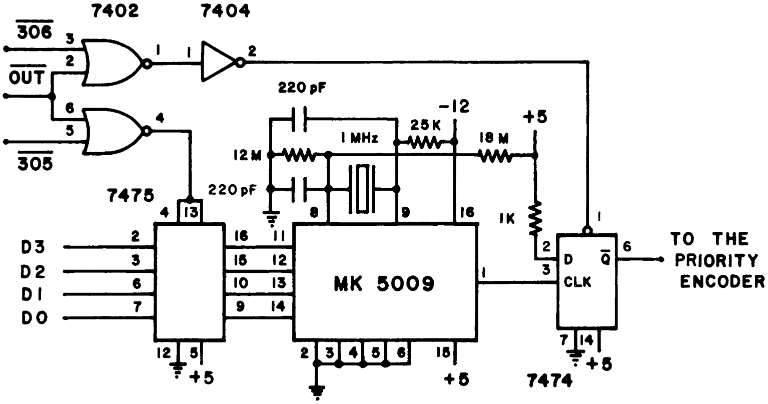


Fig. 3-2. A microcomputer-programmable real-time clock.

If we program the MK5009 for 1-kHz operation, then 23 interrupts will have to occur for a delay of 23 one-millisecond periods. For a delay of 200 seconds, 200 periods of 0.1 second must occur. The interrupt service subroutine listed in Example 3-1 keeps track of the number of time intervals that have occurred, by decrementing a count.

After the stack pointer and register pair H in Example 3-1 are loaded with R/W memory addresses, 027 (decimal 23) is saved in the memory locations assigned the symbolic addresses COUNT and

TEMP. This is the number of times that an interrupt must occur before the peripheral device is serviced. The A register is then loaded with 003 (03) and this value is output to the real-time clock latch. This value (based on Table 3-1) programs the MK5009 for 1-kHz (1 ms) operation. The second OUT instruction clears the flip-flop that is driven by the output of the MK5009 (Fig. 3-2), and the interrupt is then enabled. The 8080 then executes the MAIN TASK.

Table 3-1. Programming Inputs and Clock Frequencies Output by the MK5009 With a 1-MHz Crystal

Programming Inputs				Clock Frequency Output	Period
2^3	2^2	2^1	2^0		
0	0	0	0	1 MHz	1 μs
0	0	0	1	100 kHz	10 μs
0	0	1	0	10 kHz	100 μs
0	0	1	1	1 kHz	1 ms
0	1	0	0	100 Hz	10 ms
0	1	0	1	10 Hz	100 ms
0	1	1	0	1 Hz	1 s
0	1	1	1	0.1 Hz	10 s
1	0	0	0	0.01 Hz	100 s

The real-time clock will then interrupt the microcomputer in from 0 to 1 ms. Based on the interrupt hardware that was seen in the previous chapter, we know that a restart instruction is placed on the data bus and is written into the instruction register when the 8080 generates an interrupt acknowledge ($\overline{\text{INTA}}$) signal. Since the interrupt service subroutine in Example 3-1 starts at 000 070 (0038), we can assume that the output of the flip-flop in Fig. 3-2 is wired to the highest priority interrupt. When the 8080 is interrupted, it executes the instructions in the interrupt service subroutine RTCISS (Example 3-1).

At RTCISS, register pair H and the PSW are saved on the stack. Register pair H is then loaded with the memory address of COUNT. The content of this memory location is then decremented by 1. If the result of the DCRM instruction is 0, the 8080 executes the JZ to PSERV. This means that 23 interrupts have occurred. If 23 interrupts have not occurred, the JZ to PSERV is not executed. Instead, the PSW and register pair H are popped off of the stack. The interrupt flip-flop is then cleared (OUT 306) and the interrupt is reenabled. The 8080 then returns to the section of MAIN TASK that was interrupted.

If 23 interrupts have occurred, the 8080 jumps to PSERV. At PSERV, the 8080 increments the memory address in register pair

Example 3-1: A Real-Time Clock Interrupt Service Subroutine

```
        *000 000
START,  LXISP   /LOAD THE SP WITH A R/W
        STACK   /MEMORY ADDRESS.
        0
        LXIH    /LOAD REGISTER PAIR H WITH THE
        COUNT   /MEMORY ADDRESS ASSIGNED TO
        0       /"COUNT."
        MVIM    /SAVE A DECIMAL 23 IN THIS
        027     /MEMORY LOCATION.
        INXH    /INCREMENT THE ADDRESS IN H AND L.
        MVIM    /SAVE A DECIMAL 23 IN "TEMP."
        027
        MVIA    /THEN LOAD THE A REGISTER WITH THE
        003     /WORD THAT PROGRAMS THE MK5009
        OUT     /FOR 1-KHZ (1 MSEC) OPERATION.
        305     /OUTPUT THIS VALUE.
        OUT     /CLEAR THE INTERRUPT FLIP-FLOP.
        306
        EI      /ENABLE THE INTERRUPT
        •       /AND EXECUTE THE "MAIN TASK."
        •
        •

        *000 070
RTCISS, PUSHH   /SAVE REGISTER PAIR H,
        PUSHPSW /A, AND THE FLAGS.
        LXIH    /LOAD REGISTER PAIR H WITH THE
        COUNT   /MEMORY ADDRESS WHERE THE COUNT
        0       /IS STORED.
        DCRM    /DECREMENT THE COUNT IN MEMORY.
        JZ      /THE COUNT IS ZERO, SO SERVICE
        PSERV   /THE PERIPHERAL DEVICE (A DELAY
        0       /OF 23 MSEC HAS OCCURRED).
AGAIN,  POPPSW  /THE COUNT IS NONZERO, SO POP
        POPH    /A, THE FLAGS, AND REGISTER PAIR H.
        OUT     /THEN CLEAR THE INTERRUPT FLAG.
        306
        EI      /ENABLE THE INTERRUPT
        RET     /AND RETURN TO "MAIN TASK."

PSERV,  INXH    /A DELAY OF 23 MSEC HAS OCCURRED.
        MOVAM   /SO GET THE COUNT FROM "TEMP"
        DCXH    /AND STORE IT BACK IN
        MOVMA   /"COUNT."
        •       /THEN SERVICE THE INTERRUPT.
        •
        •
        JMP     /POP THE REGISTERS OFF THE STACK,
        AGAIN   /CLEAR THE INTERRUPT FLAG, REENABLE
        0       /THE INTERRUPT, AND RETURN TO "MAIN TASK."

COUNT,  0       /THE "WORKING COUNT" IS STORED HERE.
TEMP,   0       /THE COUNT OF DECIMAL 23 IS STORED HERE.
```

H and then moves the content of TEMP into the A register. The memory address is then decremented and the content of the A register is then saved in memory at COUNT. This reinitializes the count back to 23. The 8080 can then output a data word to a peripheral device or perform some other control function. After the peripheral device has been serviced, the 8080 jumps to AGAIN, so that the PSW and register pair H are popped off of the stack and the interrupt flip-flop is cleared. The interrupt is then reenabled and the 8080 returns to the MAIN TASK.

How much time is required to service the interrupt? If the content of COUNT is decremented to a nonzero value, only 48 μs are required. If the count is decremented to 0, at least 65 μs are required, plus the time required to actually service the peripheral device. You can now see why we should not attempt to interrupt the 8080 microcomputer any faster than once every 100 μs. If we did, there would be very little time between interrupts and, therefore, very little time left to execute the instructions in MAIN TASK. We also know that a device must never interrupt its own interrupt service subroutine. Otherwise, R/W memory will start to be filled up with return addresses.

The Hardware Real-Time Clock Solution

By adding six additional integrated circuits to the real-time clock interface (Fig. 3-3), we can increase the number of applications for the real-time clock and decrease the complexity of the interrupt service subroutine (Example 3-1). By using this hardware, the 8080 is only interrupted when the entire time interval has occurred. The 8080 no longer has to decrement a count that is stored in R/W memory in order to determine if the peripheral device should be serviced.

By using three four-bit counters in the real-time clock, up to 2^{12} counts can be counted before an interrupt occurs. This means that the real-time clock can be programmed so that the microcomputer is interrupted once every four days! Note that the latch in the upper portion of Fig. 3-3 is pulsed by OUT 305, the same signal that is used to latch the programming word for the MK5009.

To program the real-time clock, a 12-bit count has to be output to the three SN7475 latches that are wired to the SN74193 counters. After this is done, the count can be transferred to the SN74193 counters from the three latches. As you can see, the clock input of the least-significant counter is driven by the output of the MK-5009. Since the counters have been wired as down-counters, the borrow output of the most-significant counter is used to clock the D flip-flop in Fig. 3-2, which actually causes the 8080 microcomputer to be interrupted. This means that the "counter chain" (the

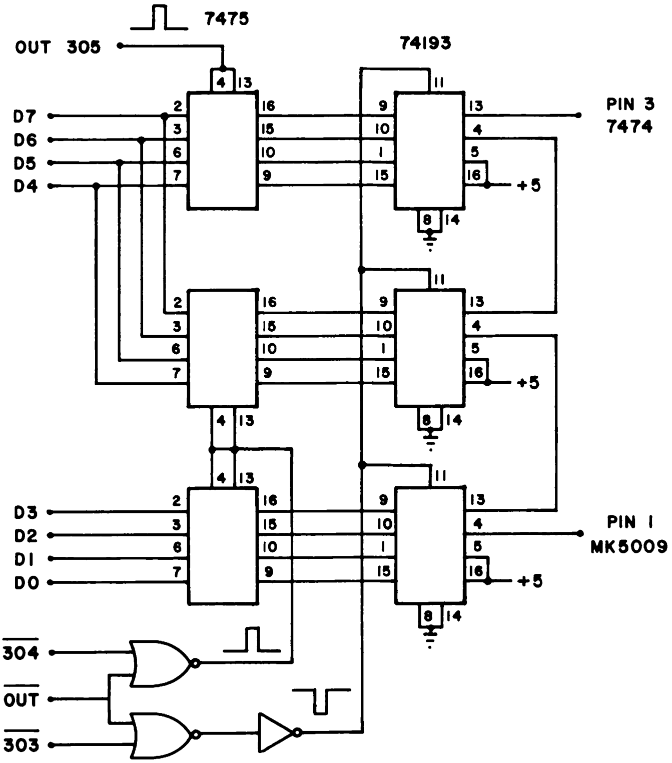


Fig. 3-3. A flexible hardware-programmable real-time clock.

three SN74193 counters) has been electrically wired between the output of the MK5009 and the input of the D flip-flop.

The Data Format for the Programmable Real-Time Clock

To program the real-time clock, a 12-bit value has to be output to the SN7475 latches, and the same value has to be transferred from the latches to the SN74193 down-counters. The MK5009 also has to be programmed for one of nine possible frequencies (periods). Two eight-bit output ports are used to latch these 16 bits of information. The arrangement of this data to program the real-time clock is shown in Table 3-2.

Based on the data format shown in Table 3-2, what values would have to be output to the real-time clock so that the 8080 microcomputer is interrupted once every 23 ms? To program the real-time clock for this time period, the values,

OUT 305	OUT 304
0000 0011	0001 0111

Table 3-2. The Data Format for the Programmable Real-Time Clock

OUT 305		OUT 304	
D_7 D_6 D_5 D_4	D_3 D_2 D_1 D_0	D_7 D_6 D_5 D_4	D_3 D_2 D_1 D_0
Most-Significant Counter	MK5009	Middle Counter	Least-Significant Counter

could be output. These values program the MK5009 for 1-kHz (1 ms) operation (data value = 0011), and the counter chain with the value 0000 0001 0111. Are there any other values that could be used to program the real-time clock? Yes, any of the values in Table 3-3 could be used to program the real-time clock for 23-ms time intervals.

Table 3-3. Values That Program the Real-Time Clock for 23-ms Interrupts

OUT 305								OUT 304							
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	1	1	0	0	0	1	0	1	1	1
0	0	0	0	0	0	1	0	1	1	1	0	0	1	1	0
1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0

The first 16-bit value in Table 3-3 programs the MK5009 for 1-kHz operation and loads the counters with decimal 23. The next value programs the MK5009 for 10-kHz operation and loads the counters with decimal 230. The last 16-bit value programs the MK5009 for 100-kHz operation and loads the counters with decimal 2300. This combination will produce a time interval with the greatest accuracy, since the maximum uncertainty in the time will be 10 μs.

What sequence of instructions could be used to program the real-time clock for 23-ms operation? Assume that the 16-bit value 00000011 00010111 must be output to the real-time clock (10 kHz, decimal count of 23). The instruction sequence in Example 3-2 will do just this.

In Example 3-2, the 8080 loads the stack pointer and then outputs the value 027 (decimal 23) to the two least-significant counters in the real-time clock (OUT 304). The A register is then loaded with the value that will program the most-significant counter *and* the MK5009. The value 00000011 loads the counter with 0000 and programs the MK5009 for 1-kHz operation (output port 305). The OUT 303 instruction then loads the three counters with the values that were previously output to the three latches, and the OUT 306 instruction clears the interrupt flag. After the interrupt is enabled (EI), the MAIN TASK is executed.

Example 3-2: Programming the Real-Time Clock for 23-ms Operation

```
        *000 000
START,  LXISP   /LOAD THE SP WITH A R/W
        STACK   /MEMORY ADDRESS.
        0
        MVIA    /THEN LOAD THE A REGISTER
        027     /WITH DECIMAL 23.
        OUT     /OUTPUT THIS VALUE TO THE
        304     /TWO LEAST SIGNIFICANT COUNTERS.
        MVIA    /THEN SET THE MOST SIGNIFICANT
        003     /COUNTER TO ZERO AND THE MK5009
        OUT     /FOR 1-KHZ (1 MSEC) OPER-
        305     /ATION.
        OUT     /LOAD THE THREE COUNTERS WITH
        303     /THE OUTPUT OF THE THREE LATCHES.
        OUT     /CLEAR THE INTERRUPT FLAG
        306     /(THE D FLIP-FLOP).
        EI      /ENABLE THE INTERRUPT
        •       /AND EXECUTE THE "MAIN TASK."
        •
        •

        *000 070
RTCISS, •       /NOW SERVICE THE PERIPHERAL
        •       /DEVICE.
        •
        OUT     /THEN LOAD THE COUNTERS AGAIN.
        303
        OUT     /CLEAR THE INTERRUPT FLAG
        306     /(THE D FLIP-FLOP).
        EI      /AND ENABLE THE INTERRUPT.
        RET     /THEN RETURN TO "MAIN TASK."
```

An interrupt will occur in 23 ms. When it does, the 8080 services the peripheral device. Because of the addition of the three counters to the real-time clock, the instructions in the interrupt service subroutine do not have to decrement a count. This means that the peripheral device is serviced *every* time an interrupt occurs. After the peripheral is serviced, the OUT 303 instruction transfers the count stored in the three latches to the SN74193 counters. The OUT 306 instruction clears the interrupt flag, and then the interrupt is reenabled and the 8080 returns to the section of MAIN TASK that was being executed when the interrupt occurred.

A Comparison of the Hardware- and Software-Intensive Real-Time Clocks

In the software-intensive real-time clock, a count has to be stored in R/W memory. Each time an interrupt occurs, this count has to be decremented. If the count is decremented to a nonzero value, the peripheral device is not serviced. If the count is decremented to 0, the count has to be reloaded (transferred from one memory location to another), and then the peripheral device has to be serviced.

The software-intensive real-time clock requires very little hardware; only five integrated circuits, assuming that the appropriate device decoders and interrupt hardware already exist. The main disadvantage of this technique is the time required to service the interrupts and the peripheral device. To service the peripheral once, 23 interrupts are generated by the real-time clock. This means that $(22 \times 48\ \mu s) + 65\ \mu s$, or 1.12 ms, are required to service the interrupt every 23 ms, plus the time required to actually service the peripheral device.

The hardware-intensive method requires eight additional integrated circuits. However, since only one interrupt occurs every 23 ms, far less time is required for the real-time clock "housekeeping" instructions (reprogramming the counters and clearing the interrupt flip-flop). In fact, only 17 μs are required for the hardware-intensive real-time clock housekeeping instruction. For applications where the 8080 microcomputer must perform tasks as quickly as possible, the hardware-intensive real-time clock will keep the time required to service a peripheral device to a minimum.

Hardware-Intensive Real-Time Clock Alternatives

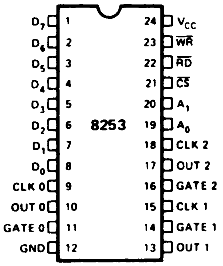
A number of semiconductor manufacturers have realized the desirability of software-programmable counters and timers. One of these devices, the Intel 8253 programmable interval timer, has three independent 16-bit counters. Each counter has a clock input, a counter output, and a control line that can be used to start and stop the counter. The pin configuration and block diagrams for this integrated circuit are shown in Fig. 3-4.

For interrupt applications, an output (or outputs) of the counter(s) (OUT 0, OUT 1, or OUT 2) can be wired to any of the priority interrupt interfaces described in the previous chapter. Since the counters in the integrated circuit can operate from dc to 2 MHz, the clock inputs (CLK 0, CLK 1, and CLK 2) can be wired to the TTL clock associated with the 8080 clock generator integrated circuit (8224), or the CLK (OUT) output of the 8085.

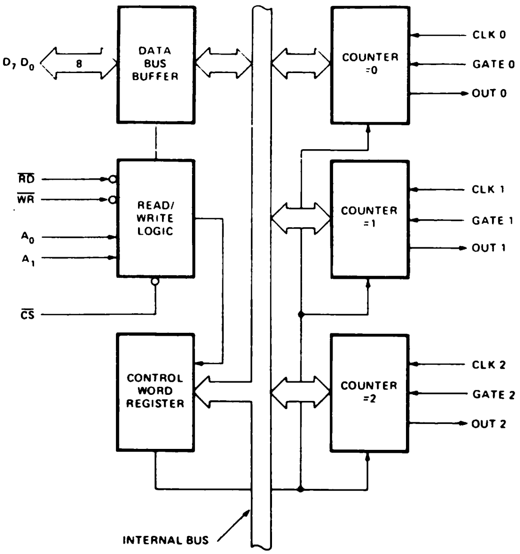
If this is done, a single counter can produce a delay of from 0.5 to 32,768 μs. However, if the output of one counter is used to clock another counter, delays of up to 2^{31} μs can be obtained; almost 18 minutes. By adding a third 16-bit counter to the counter chain, delays of over 407 days can be generated.

One outstanding feature of the 8253 is the fact that three different time delays can be programmed into it. This means that for three peripheral devices, a single timer can be dedicated to each device. This is particularly useful if the devices must be serviced at irregu-

PIN CONFIGURATION


D_7 1
D_6 2
D_5 3
D_4 4
D_3 5
D_2 6
D_1 7
D_0 8
CLK 0 9
OUT 0 10
GATE 0 11
GND 12
8253
24 V_{CC}
23 $\overline{WR}$
22 $\overline{RD}$
21 $\overline{CS}$
20 A_1
19 A_0
18 CLK 2
17 OUT 2
16 GATE 2
15 CLK 1
14 GATE 1
13 OUT 1

BLOCK DIAGRAM


D_7 D_0
8
DATA BUS BUFFER
$\overline{RD}$
$\overline{WR}$
A_0
A_1
READ/ WRITE LOGIC
$\overline{CS}$
CONTROL WORD REGISTER
INTERNAL BUS
COUNTER =0
CLK 0
GATE 0
OUT 0
COUNTER =1
CLK 1
GATE 1
OUT 1
COUNTER =2
CLK 2
GATE 2
OUT 2

PIN NAMES

D_7 D_0	DATA BUS (8 BIT)
CLK N	COUNTER CLOCK INPUTS
GATE N	COUNTER GATE INPUTS
OUT N	COUNTER OUTPUTS
$\overline{RD}$	READ COUNTER
$\overline{WR}$	WRITE COMMAND OR DATA
$\overline{CS}$	CHIP SELECT
A_0 A_1	COUNTER SELECT
V_{CC}	+5 VOLTS
GND	GROUND

Courtesy Intel Corp.

Fig. 3-4. The Intel 8253 programmable interval timer.

lar intervals, for instance, 233 μs, 500 μs, and 1.895 ms. To do this with the real-time clock based on the MK5009, we would have to build three programmable real-time clocks.

However, there is a notable disadvantage in using the 8253 for very complex timing problems. Each time the timer produces an interrupt, or "times-out," the timer must be reloaded with a 16-bit count. This is done by executing two OUT instructions, two MOV-type instructions, or an SHLD instruction. The type of instruction(s) used will be determined by the interface (accumulator I/O or memory-mapped I/O) used with the 8253. With the real-time clock based on the MK5009, a single OUT instruction causes a 12-bit count to be loaded into the three SN74193 counters. For additional information on the 8253, refer to Intel's *Component Data Catalog*[1].

Conclusion

As we noted previously, it may not be time-efficient to have the 8080 microcomputer continuously polling peripheral devices to determine whether or not they need service. Instead, by using interrupts, peripheral devices can notify the 8080 when they need service.

At the same time, it may not be efficient to have the microcomputer execute time delay subroutines because it may have more important tasks to perform. For this reason, real-time clocks are used. They relieve the microcomputer of executing time delay subroutines.

A TIME-OF-DAY CLOCK

By using a real-time clock, a program can be written that keeps time in hours, minutes, and seconds. This program performs the same function as a hardware clock. Our first *time-of-day clock* example will keep time in a 24-hour format. Therefore, the time 1:32 p.m. will really be represented as 13:32 when the time-of-day clock program is executed. We already know (Table 3-1) that the MK5009 can be programmed to interrupt the 8080 once every second. When the microcomputer is interrupted, it will have to increment a BCD count, stored in memory, which represents the number of seconds. If this number is incremented to BCD 60, then the number of seconds must be set to 0 and the number of minutes must be incremented by 1. If the number of BCD minutes is now equal to 60, then the number of minutes must be set to 0 and the number of BCD hours must be incremented by 1. If the number of BCD hours is now equal to 24, then the number of hours must be set to 0. Since we are using packed BCD values, only three R/W memory locations are required to store the time. Another three memory locations are required to store the maximum number of hours, minutes, and seconds (24, 60, and 60) that are used to indicate when the next more-significant number should be incremented by 1. All of these operations are performed by the program listed in Example 3-3. Since the MK5009 can produce a single pulse every second, there is no need to use the three SN74193 programmable down-counters (Fig. 3-3). Therefore, the output of the MK5009 can be rewired to the D flip-flop that is wired to the priority interrupt interface.

The time-of-day clock program starts at 002 000 (0200). When the 8080 begins to execute this program, it loads the SP because the interrupt hardware will write a restart instruction into the instruction register when the real-time clock times-out. The A register is then loaded with 006, which is output to the MK5009. This programs the MK5009 for 1-Hz (1 s) operation. Register pair H is then loaded with the memory address where the time is stored. The B register is then loaded with 003 (03), so that the ZERO loop is executed three times. At ZERO, the memory location addressed by register pair H is set to 0. The memory address in register pair H is then incremented and the content of the B register is decremented. This loop causes BCD 00 to be saved in

Example 3-3: An Interrupt-Driven Time-of-Day Clock Program

```
/THIS IS THE INTERRUPT SERVICE SUBROUTINE. THE IN-
/TERRUPT HARDWARE GENERATES A RST4 INSTRUCTION.

        *000 040
CLOCK,  PUSHPSW /WHEN THE INTERRUPT OCCURS, SAVE
        PUSHB   /ALL OF THE REGISTERS ON THE STACK.
        PUSHD
        PUSHH
        JMP     /CONTINUE TO SERVICE THE INTERRUPT
        ISCNT   /AT "INTERRUPT SERVICE—CONTINUE"
        0       /(ISCNT).

/THIS IS A TIME-OF-DAY CLOCK PROGRAM
/(THIS IS THE START OF THE "MAIN TASK").

        *002 000
START,  LXISP   /LOAD THE STACK POINTER BECAUSE
        STACK   /AN INTERRUPT WILL CAUSE A RETURN AD-
        0       /DRESS TO BE SAVED ON THE STACK.
        MVIA    /LOAD THE A REGISTER WITH THE
        006     /WORD THAT WILL PROGRAM THE
        OUT     /MK5009 FOR 1-HZ OPERATION.
        305     /AND THEN LATCH THE WORD OUT.
        LXIH    /LOAD REGISTER PAIR H WITH THE AD-
        CURTIM  /DRESS WHERE THE CURRENT TIME (CURTIM)
        0       /IS STORED IN MEMORY (BCD FORMAT).
        MVIB    /LOAD THE B REGISTER WITH THE
        003     /NUMBER OF LOCATIONS USED BY "CURTIM."
ZERO,   MVIM    /LOAD THE MEMORY LOCATION WITH 000.
        000
        INXH    /INCREMENT THE MEMORY ADDRESS.
        DCRB    /DECREMENT THE WORD COUNT.
        JNZ     /IF THE  COUNT IS NONZERO,
        ZERO    /SET ANOTHER MEMORY LOCATION
        0       /TO 000.
        OUT     /CLEAR THE MK5009'S INTERRUPT
        306     /FLAG.
        EI      /ENABLE THE INTERRUPT AND CONTINUE
        •       /TO EXECUTE THE "MAIN TASK."
        •
        •

/THE 8080 JUMPS TO  "ISCNT" TO CONTINUE SERVICING
/THE INTERRUPT THAT IS GENERATED ONCE PER SECOND.

ISCNT,  LXIH    /LOAD REGISTER PAIR H WITH THE ADDRESS
        TOPVAL  /OF THE TABLE THAT CONTAINS THE MAXIMUM
        0       /NUMBER OF SECONDS, MINUTES, AND HOURS.
        LXID    /THEN LOAD REGISTER PAIR
        CURTIM  /D WITH THE ADDRESS WHERE THE
        0       /CURRENT TIME IS STORED.
        MVIB    /LOAD THE B REGISTER WITH THE NUMBER
        003     /OF MEMORY LOCATIONS INVOLVED.
```

```
UPONE,   LDAXD    /GET THE NUMBER.
         ADI      /ADD ONE TO IT.
         001
         DAA      /AND ADJUST THE RESULT.
         STAXD    /SAVE THE NEW TIME.
         CMPM     /IS IT TOO LARGE?
         JZ       /YES, THEN SET THIS
         NEXT     /VALUE TO ZERO AND INCREMENT THE
         0        /NEXT TWO DIGITS OF THE TIME.
MIDNGT,  LXIH     /THEN DISPLAY THE CURRENT TIME.
         CURTIM
         0
         MOVAM    /GET THE BCD NUMBER OF SECONDS
         OUT      /AND OUTPUT THE NUMBER TO THE
         002      /SEVEN-SEGMENT DISPLAYS.
         INXH     /INCREMENT THE MEMORY ADDRESS.
         MOVAM    /GET THE BCD NUMBER OF MINUTES
         OUT      /AND OUTPUT THE NUMBER TO THE
         000      /SEVEN-SEGMENT DISPLAYS.
         INXH     /INCREMENT THE MEMORY ADDRESS.
         MOVAM    /GET THE BCD NUMBER OF HOURS
         OUT      /AND OUTPUT THE NUMBER TO
         001      /THE SEVEN-SEGMENT DISPLAYS.
         POPH     /THE TIME IS VALID, SO GET ALL
         POPD     /OF THE REGISTERS OFF OF THE STACK.
         POPB
         POPPSW
         OUT      /NOW CLEAR THE INTERRUPT FLIP-
         306      /FLOP WIRED TO THE MK5009.
         EI       /ENABLE THE INTERRUPT
         RET      /AND RETURN TO THE "MAIN TASK."

NEXT,    MVIA     /SET THE A REGISTER TO 000
         000      /(BCD AND HEX 00).
         STAXD    /THEN SAVE IT IN THE CURRENT TIME.
         INXH     /INCREMENT THE TABLE ADDRESS.
         INXD     /INCREMENT THE CURRENT TIME ADDRESS.
         DCRB     /DECREMENT THE NUMBER OF LOCATIONS.
         JNZ      /THE COUNT IS NONZERO, SO THE
         UPONE    /NEXT CONSECUTIVE MEMORY LOCATION
         0        /CAN BE INCREMENTED.
         JMP      /THE HOURS DIGIT IS EQUAL TO 24,
         MIDNGT   /SO IT IS MIDNIGHT. DO NOT IN-
         0        /CREMENT ANY MEMORY LOCATIONS.

CURTIM,  000      /THE NUMBER OF BCD SECONDS HERE.
         000      /THE NUMBER OF BCD MINUTES HERE.
HOURS,   000      /THE NUMBER OF BCD HOURS HERE.

TOPVAL,  140      /THE MAX. NUMBER OF BCD SECONDS = 60.
         140      /THE MAX. NUMBER OF BCD MINUTES = 60.
         044      /THE MAX. NUMBER OF BCD HOURS = 24.
```

three consecutive memory locations, starting at CURTIM. The result is that the time is set to 00:00:00. After the 8080 initializes the time, the interrupt flip-flop in the real-time clock interface is cleared and the interrupt is enabled. The 8080 then executes MAIN TASK.

When the real-time clock interrupts the 8080, the 8080 is vectored to 000 040 (0020). Starting at this memory address, the interrupt service subroutine for the real-time clock must be stored in memory. Starting at CLOCK (000 040, 0020), *all* of the 8080 general-purpose registers and the flags are saved on the stack. The 8080 then jumps to the remainder of the interrupt service subroutine at ISCNT. The interrupt service subroutine was written like this so that another peripheral device can use the RST5 instruction as an interrupt instruction.

At ISCNT, register pair H is loaded with the memory address where the "maximum" number of BCD seconds is stored. Register pair D is loaded with the memory address where the number of BCD seconds in the "current" time is stored, and the B register is loaded with the number of memory locations that are used to store the current time. At UPONE, the number of seconds in the current time is loaded into the A register from memory. One is added to this number and the result is decimally adjusted (this can be done because the 8080 is operating on packed BCD numbers). The result is stored in the memory location addressed by register pair D. This same value is then compared to the maximum number of BCD seconds that is stored in the TOPVAL table. If the number of seconds in the current time is not equal to BCD 60, the instructions at MIDNGT are executed.

At MIDNGT, register pair H is loaded with the memory address where the number of seconds in the current time is stored. The BCD number of seconds is then output to output port 002, the number of minutes to output port 000, and the number of hours to output port 001. These output ports are equipped with latches and seven-segment light-emitting-diode (LED) displays, so the current time is now being displayed. For further information about output ports equipped with LED displays, refer to Chapter 7 of *8080/8085 Software Design—Book 1*[2]. After the time is displayed, all the registers are popped off the stack, the interrupt flag is cleared, the interrupt is reenabled, and the 8080 returns to MAIN TASK.

If the number of BCD seconds in the current time is equal to 60, the JZ to NEXT is executed (just before MIDNGT). At NEXT, the number of seconds in the current time is set to 0. The memory addresses in register pairs D and H are then incremented and the count in the B register is decremented from 003 (03) to 002 (02). Since this is a nonzero result, the 8080 jumps to UPONE.

If the number of seconds was incremented to BCD 60, the number of minutes now has to be incremented by 1. If the number of minutes is now incremented to 60, the 8080 sets the number of minutes to 0 and then increments the number of hours. If this causes the number of hours to be incremented to 24, then the number of hours is set to 0. After the 8080 does this, it jumps to MIDNGT, where the time is displayed and all of the registers are popped off of the stack. The 8080 then clears the interrupt flag, reenables the interrupt, and returns to the MAIN TASK.

Initializing the Time-of-Day Clock With a Fixed Time

When the time-of-day clock program is started at 002 000 (0200), the time is set to 00:00:00. Suppose you load this program into your 8080 microcomputer at 10:15 a.m. Could you start the program and have the time 10:15 displayed? Yes, the 8080 could display this time. One method that you could use is listed in Example 3-4.

Example 3-4: Saving the Time 10:15:00 in R/W Memory

```
        *002 000
START,  LXISP   /LOAD THE STACK POINTER BECAUSE
        STACK   /AN INTERRUPT WILL CAUSE A RETURN AD-
        0       /DRESS TO BE SAVED ON THE STACK.
        MVIA    /LOAD THE A REGISTER WITH THE VALUE
        006     /THAT WILL PROGRAM THE MK5009 FOR
        OUT     /1-HZ OPERATION.
        305
        LXIH    /LOAD REGISTER PAIR H WITH THE AD-
        CURTIM  /DRESS WHERE THE CURRENT TIME (CURTIM)
        0       /IS STORED IN MEMORY (BCD FORMAT).
        MVIM    /THEN SET THE NUMBER OF SECONDS TO 00.
        000
        INXH    /INCREMENT THE ADDRESS TO THE MINUTES.
        MVIM    /THEN SET THE NUMBER OF MINUTES TO 15.
        025
        INXH    /INCREMENT THE ADDRESS TO THE HOURS.
        MVIM    /THEN SET THE NUMBER OF HOURS TO 10.
        020
        OUT     /CLEAR THE MK5009'S INTERRUPT
        306     /FLAG.
        EI      /ENABLE THE INTERRUPT
        •       /AND EXECUTE THE "MAIN TASK."
        •
        •
```

In this program, a fixed time is simply stored in the current time R/W memory locations, using three MVIM instructions. This method is not particularly good if the program has to be stored in read-only memory, and the time used to program the time-of-day clock must be changed.

Of course, we can also write a teletypewriter-oriented program that enables us to enter the actual time. If this is done, the 8080 must check the time that is entered to see if the number of hours specified is greater than 23. It must also ensure that the number of minutes or seconds specified is not greater than 59. The program listed in Example 3-5 lets you program the time-of-day clock with a time that is entered on the teletypewriter.

Example 3-5: Entering a Time Into the Microcomputer Using a Teletypewriter

```
/THIS IS A TIME-OF-DAY CLOCK PROGRAM
/(THIS IS THE START OF THE "MAIN TASK").

          *002 000
START,    LXISP    /LOAD THE STACK POINTER BECAUSE
          STACK    /AN INTERRUPT WILL CAUSE A RETURN AD-
          0        /DRESS TO BE SAVED ON THE STACK.
          MVIA     /LOAD THE A REGISTER WITH THE
          006      /WORD THAT WILL PROGRAM THE
          OUT      /MK5009 FOR 1-HZ OPERATION.
          305      /AND THEN LATCH THE WORD OUT.
          LXIH     /LOAD REGISTER PAIR H WITH THE
          HOURS    /MEMORY ADDRESS WHERE THE NUMBER OF
          0        /HOURS CAN BE STORED.
          MVID     /THEN LOAD THE D REGISTER WITH THE
          003      /NUMBERS OF "WORDS" THAT CAN BE ENTERED.
          CALL     /PRINT A CARRIAGE RETURN AND A
          CRLF     /LINE FEED ON THE CRT OR
          0        /TELETYPEWRITER.
NXTDIG,   CALL     /GET A TWO-DIGIT BCD NUMBER AND
          TIMIN    /RETURN WITH THE TWO-DIGIT PACKED
          0        /BCD WORD IN THE A REGISTER.
          MOVMA    /SAVE THE NUMBER IN MEMORY.
          DCXH     /DECREMENT THE MEMORY ADDRESS.
          DCRD     /AND DECREMENT THE DIGIT COUNT.
          JZ       /THE COUNT IS ZERO, SO SEE IF A
          CHECK    /VALID TIME WAS ENTERED.
          0
          MVIA     /OTHERWISE, PRINT A COLON AFTER
          072      /THE TWO-DIGIT NUMBER JUST ENTERED.
          CALL
          TTYOUT
          0
          JMP
          NXTDIG   /THEN GET ANOTHER TWO-DIGIT NUMBER.
          0
CHECK,    LXID     /NOW COMPARE THE TIME ENTERED TO
          TOPVAL   /THE MAXIMUM ALLOWABLE VALUES.
          0
          INXH     /INCREMENT H AND L TO THE SECONDS.
          MVIC     /LOAD THE C REGISTER WITH THE NUMBER
          003      /OF VALUES TO BE CHECKED.
```

```
AGAIN,  LDAXD   /GET A VALUE FROM THE "TOPVAL" TABLE.
        DCRA    /DECREMENT THE NUMBER BY ONE.
        CMPM    /COMPARE IT TO A NUMBER THAT WAS ENTERED.
        JNC     /THE NUMBER ENTERED IS LESS THAN
        CNEXT   /THE TABLE ENTRY, SO TEST THE NEXT
        0       /ENTRY THAT WAS TYPED IN.
        MVIA    /THE NUMBER ENTERED WAS TOO LARGE,
        277     /SO PRINT A QUESTION MARK AND
        CALL    /LET THE USER TRY AGAIN.
        TTYOUT
        0
        JMP
        START
        0
CNEXT,  INXH    /INCREMENT THE MEMORY ADDRESSES
        INXD    /TO THE NEXT TWO ENTRIES.
        DCRC    /ALL THREE COMPARED YET?
        JNZ     /NO, THEN COMPARE THE NEXT TWO
        AGAIN   /ENTRIES.
        0
        CALL    /THE TIME ENTERED IS VALID, SO
        TTYIN   /WAIT FOR A KEY TO BE PRESSED BEFORE
        0       /"STARTING" THE CLOCK.
        OUT     /CLEAR THE MK5009'S INTERRUPT
        306     /FLAG.
        EI      /ENABLE THE INTERRUPT AND CONTINUE
        •       /TO EXECUTE THE "MAIN TASK."
        •
        •

TIMIN,  CALL    /GET A VALID BCD NUMBER AND RETURN
        BCDIN   /WITH IT IN THE A REGISTER.
        0
        RLC     /THEN ROTATE IT INTO THE
        RLC     /FOUR MSB'S.
        RLC
        RLC
        MOVCA   /AND SAVE IT IN C.
        CALL    /GET THE NEXT DIGIT.
        BCDIN
        0
        ADDC    /ADD THE PREVIOUS NUMBER.
        RET     /AND RETURN WITH THE RESULT IN A.

BCDIN,  CALL    /GET A CHARACTER FROM THE
        TTYIN   /TELETYPEWRITER.
        0
        CPI     /IS IT LESS THAN ASCII 0?
        060
        JC      /YES, THEN IGNORE IT.
        BCDIN
        0
        CPI     /IS IT GREATER THAN ASCII 9?
        072
        JNC     /YES, THEN IGNORE IT.
        BCDIN
        0
        ANI      /SAVE ONLY THE FOUR LSB'S.
        017
        RET      /AND RETURN WITH IT IN A.

CRLF,   MVIA     /LOAD THE A REGISTER WITH THE ASCII
        215      /VALUE FOR THE CARRIAGE RETURN CHARACTER
        CALL     /AND PRINT IT.
        TTYOUT
        0
        MVIA     /THEN LOAD THE A REGISTER WITH THE
        212      /VALUE FOR THE LINE FEED CHARACTER
        JMP      /AND PRINT IT.
        TTYOUT
        0

TTYIN,  IN       /INPUT THE UART'S STATUS BITS.
        001
        ANI      /SAVE ONLY THE RECEIVER'S FLAG.
        001      /IF A=001, A KEY IS PRESSED.
        JZ       /IF A=000, NO KEY IS PRESSED.
        TTYIN    /SO KEEP WAITING FOR A KEY
        0        /TO BE PRESSED.
        IN       /A KEY IS PRESSED, SO INPUT THE
        000      /ASCII CHARACTER INTO THE A REGISTER.
        ANI      /THEN SET THE PARITY BIT (D7)
        177      /TO ZERO (177 = HEX 7F).
TTYOUT, MOVBA    /SAVE THE CHARACTER IN B.
TTYO,   IN       /INPUT THE UART'S STATUS WORD.
        001
        ANI      /SAVE ONLY THE TRANSMITTER'S FLAG.
        004      /IF A=004, THE TRANSMITTER (PRINTER) IS READY.
        JZ       /IF A=000, THE TRANSMITTER (PRINTER) IS BUSY.
        TTYO     /SO KEEP WAITING FOR THE TRANSMITTER
        0        /(PRINTER) TO FINISH, BEFORE THE
        MOVAB    /CONTENT OF THE A REGISTER CAN BE PRINTED.
        OUT      /AFTER THE CHARACTER IS MOVED FROM
        000      /B TO A, OUTPUT IT TO THE UART.
        RET      /RETURN WITH THE CHARACTER STILL IN A.
```

After the stack pointer is loaded in Example 3-5, the MK5009 is programmed for 1-Hz (1 s) operation, and then register pair H is loaded with the memory address where the number of packed BCD hours that will be used to program the clock will be stored. The D register is then loaded with 3, because three two-digit numbers must be entered. A carriage return and line feed are then printed on the teletypewriter before the 8080 calls the TIMIN subroutine at NXTDIG. The TIMIN subroutine inputs two ASCII numeric characters and packs them into one eight-bit register. If nonnumeric characters are entered, the TIMIN subroutine ignores them. The 8080 will only return from the TIMIN subroutine when two numeric characters have been entered and packed into the

A register. When the 8080 returns from the TIMIN subroutine the first time, the BCD number of hours in the A register is saved in the memory location addressed by register pair H. The memory address and count are then decremented by 1. If the count is nonzero, a colon is printed on the teletypewriter and the 8080 jumps back to NXTDIG so that the number of minutes can be entered. When the number of hours, minutes, and seconds have been entered (in that order), the 8080 jumps to CHECK.

Now that the six-digit time has been entered, the 8080 has to determine if it is a valid time. This means that the number of hours cannot be greater than 23, and the number of minutes or seconds cannot be greater than 59. At CHECK, register pair D is loaded with the address where the nonvalid number of seconds is stored in memory. This same table (TOPVAL) is also used when an interrupt occurs and the 8080 increments the time and compares the new time to the entries in this table. After register pair D is loaded, the R/W memory address in register pair H is incremented, so that register pair H addresses the memory location where the number of seconds that were input were stored. The C register is then loaded with the number of memory locations used to store the time.

After these registers are initialized, the 8080 reads a value from the TOPVAL table and decrements it by 1. This value is then compared to the number of seconds that the user entered on the teletypewriter. If the value entered is less than or equal to the decremented TOPVAL number, the JNC to CNEXT is executed. At CNEXT, the memory addresses in register pairs D and H are incremented and the count in the C register is decremented. If three comparisons have not been made yet, the JNZ to AGAIN is executed. If the number entered is not a valid number of seconds, the JNC to CNEXT is not executed. Instead, a question mark is printed on the teletypewriter. The 8080 then jumps back to START so that a new, and hopefully valid, six-digit time can be entered.

If a valid six-digit time has been entered, the JNZ to AGAIN (just after CNEXT) is not executed. Instead, the TTYIN subroutine is called. Why is this done? If you entered a time such as 12:05:36, the time is accurate to one second. However, how long will it take you to enter this six-digit time into the microcomputer using a teletypewriter? Four or five seconds? This means that to program the time-of-day clock with the time 12:05:36, you would have to start entering the time at perhaps 12:05:31, so that by the time you had finished entering the time, the entered time is equal to the actual or real time. To simplify this synchronization task, the 8080 calls the TTYIN subroutine once a valid six-digit time has been entered.

This means that to program the clock with the time 12:05:36, you might start entering this time into the microcomputer at 12:05:01. Once this time has been entered, you wait for your watch to read 12:05:36. When it does, you press any printing teletypewriter key. When you do this the 8080 inputs the ASCII character into the A register and returns from the subroutine. More importantly, however, the 8080 then clears the interrupt flag and enables the interrupt. The reason that the TTYIN subroutine was called was not so that a character could be entered into the microcomputer. Instead, TTYIN was called so that the 8080 waits in a loop until the entered time is equal to the real time. When the two times are equal, we press a key and the 8080 gets out of the loop and returns from the subroutine. By combining Examples 3-3 and 3-5, you can program your 8080 microcomputer so that it operates as a programmable time-of-day clock.

Adding an A.M./P.M. Indicator to the Time-of-Day Clock

There are a number of improvements and modifications that we can make to the time-of-day clock software examples. One easy modification would be to change the clock from a 24-hour format to a 12-hour format, while adding an a.m./p.m. indicator. To store the a.m./p.m. indicator, we will use a single R/W memory location. If the content of this memory location is 0, the time stored in CURTIM is a.m., and if the a.m./p.m. indicator is 377 (FF), then the time in CURTIM is p.m. If desired, a single data bit could be latched out to an LED to indicate a.m. or p.m.

The time-of-day clock software that uses an a.m./p.m. indicator is listed in Example 3-6. To keep the example short, we have not included the teletypewriter-oriented programming instructions. The modifications for the addition of the a.m./p.m. indicator are very simple. At the beginning of the program (START), the data byte of the MVIB instruction has been changed from 003 to 004 (03 to 04). This means that four memory locations will be set to 0; the current time (three memory locations) and the a.m./p.m. indicator (one memory location). This means that the a.m./p.m. indicator is stored in R/W memory just after the number of hours in the current time.

Unlike the previous time-of-day clock subroutines, the UPONE loop is only executed when the number of seconds or minutes is being incremented. When the number of hours must be incremented, the instructions just after NEXT are executed. The number of hours is read from memory, incremented, and stored back in memory. However, if the time has just been incremented from 11:59:59 to 12:00:00, the a.m./p.m. indicator must be changed. The CPI 022 and JZ CAMPM check for this condition. Also, there is the possibility

Example 3-6: An A.M./P.M. Indicator for the Time-of-Day Clock

```
/THIS IS THE INTERRUPT SERVICE SUBROUTINE. THE IN-
/TERRUPT HARDWARE GENERATES A RST4 INSTRUCTION.

        *000 040
CLOCK,  PUSHPSW /WHEN THE INTERRUPT OCCURS, SAVE
        PUSHB   /ALL OF THE REGISTERS ON THE STACK.
        PUSHD
        PUSHH
        JMP     /CONTINUE TO SERVICE THE INTERRUPT
        ISCNT   /AT "INTERRUPT SERVICE—CONTINUE"
        0       /(ISCNT).

/THIS IS A TIME-OF-DAY CLOCK PROGRAM
/(THIS IS THE START OF THE "MAIN TASK").

        *002 000
START,  LXISP   /LOAD THE STACK POINTER BECAUSE
        STACK   /AN INTERRUPT WILL CAUSE A RETURN AD-
        0       /DRESS TO BE SAVED ON THE STACK.
        MVIA    /LOAD THE A REGISTER WITH THE
        006     /WORD THAT WILL PROGRAM THE
        OUT     /MK5009 FOR 1-HZ OPERATION.
        305     /AND THEN LATCH THE WORD OUT.
        LXIH    /LOAD REGISTER PAIR H WITH THE
        000     /PACKED BCD MINUTES AND SECONDS.
        000
        SHLD    /STORE THIS TIME IN R/W
        CURTIM  /MEMORY.
        0
        MVIL    /THEN SET THE L REGISTER TO
        022     /BCD 12, SO THAT THE HOURS ARE
        SHLD    /INITIALIZED TO 12. THE AM/PM
        HOURS   /INDICATOR IS INITIALIZED TO
        0       /ZERO.
        OUT     /CLEAR THE MK5009'S INTERRUPT
        306     /FLAG.
        EI      /ENABLE THE INTERRUPT AND CONTINUE
        •       /TO EXECUTE THE "MAIN TASK."
        •
        •

/THE 8080 JUMPS TO "ISCNT" TO CONTINUE SERVICING
/THE INTERRUPT THAT IS GENERATED ONCE PER SECOND.

ISCNT,  LXIH    /LOAD REGISTER PAIR H WITH THE ADDRESS
        TOPVAL  /OF THE TABLE THAT CONTAINS THE MAXIMUM
        0       /NUMBER OF SECONDS, MINUTES, AND HOURS.
        LXID    /THEN LOAD REGISTER PAIR
        CURTIM  /D WITH THE ADDRESS WHERE THE
        0       /CURRENT TIME IS STORED.
        MVIB    /NUMBER OF MEMORY LOCATIONS USED
        002     /BY THE SECONDS AND MINUTES.
```

```
UPONE,  LDAXD   /GET THE NUMBER.
        ADI     /ADD ONE TO IT.
        001
        DAA     /AND ADJUST THE RESULT.
        STAXD   /SAVE THE NEW TIME.
        CMPM    /IS IT TOO LARGE?
        JZ      /YES, THEN SET THIS
        NEXT    /VALUE TO ZERO AND INCREMENT THE
        0       /NEXT TWO DIGITS OF THE TIME.
MIDNGT, LXIH    /NOW DISPLAY THE CURRENT TIME.
        CURTIM
        0
        MOVAM   /GET THE BCD NUMBER OF SECONDS.
        OUT     /AND OUTPUT THE NUMBER TO THE
        002     /SEVEN-SEGMENT DISPLAYS.
        INXH    /INCREMENT THE MEMORY ADDRESS.
        MOVAM   /GET THE BCD NUMBER OF MINUTES.
        OUT     /AND OUTPUT THE NUMBER TO THE
        000     /SEVEN-SEGMENT DISPLAYS.
        INXH    /INCREMENT THE MEMORY ADDRESS.
        MOVAM   /GET THE BCD NUMBER OF HOURS.
        OUT     /AND OUTPUT THE NUMBER TO THE
        001     /SEVEN-SEGMENT DISPLAYS.
        POPH    /THE TIME IS VALID, SO GET ALL
        POPD    /OF THE REGISTERS OFF OF THE STACK.
        POPB
        POPPSW
        OUT     /NOW CLEAR THE INTERRUPT FLIP-
        306     /FLOP WIRED TO THE MK5009.
        EI      /ENABLE THE INTERRUPT.
        RET     /AND RETURN.

NEXT,   MVIA    /SET THE A REGISTER TO 000
        000     /(BCD AND HEX 00).
        STAXD   /THEN SAVE IT IN THE CURRENT TIME.
        INXH    /INCREMENT THE TABLE ADDRESS.
        INXD    /INCREMENT THE CURRENT TIME ADDRESS.
        DCRB    /DECREMENT THE NUMBER OF LOCATIONS.
        JNZ     /THE COUNT IS NONZERO, SO THE
        UPONE   /NEXT CONSECUTIVE MEMORY LOCATION
        0       /CAN BE INCREMENTED.
        LDAXD   /NOW GET THE NUMBER OF HOURS.
        ADI     /ADD ONE TO IT.
        001
        DAA     /ADJUST THE RESULT AND
        STAXD   /SAVE IT IN MEMORY.
        CPI     /HAS THE TIME JUST CHANGED FROM
        022     /11:59:59 TO 12:00:00?
        JZ      /YES, SO THE AM/PM INDICATOR
        CAMPM   /SHOULD BE CHANGED.
        0
        CMPM    /12:59:59 INCREMENTED TO 13:00:00?
        JNZ     /NO, THEN THE NUMBER OF HOURS
        MIDNGT  /IS VALID, RETURN FROM THE
        0        /INTERRUPT.
        MVIA     /YES, THE TIME IS 13:00:00, SO
        001      /CHANGE THE TIME TO 01:00:00.
        STAXD    /SAVE THIS TIME IN MEMORY
        JMP      /AND RETURN FROM THE INTERRUPT.
        MIDNGT
        0

CAMPM,  INXD     /THE TIME IS 12:00:00, SO INCREMENT
        LDAXD    /TO THE AM/PM INDICATOR, GET IT,
        CMA      /COMPLEMENT IT, AND SAVE
        STAXD    /IT BACK IN MEMORY.
        JMP      /NOW THAT THE AM/PM INDICATOR
        MIDNGT   /HAS BEEN CHANGED, DISPLAY THE
        0        /TIME AND RETURN FROM THE INTERRUPT.

CURTIM, 000      /THE NUMBER OF BCD SECONDS HERE.
        000      /THE NUMBER OF BCD MINUTES HERE.
HOURS,  000      /THE NUMBER OF BCD HOURS HERE.
AMPM,   000      /THE AM/PM INDICATOR HERE (AM =
                 /000, PM = 377).

TOPVAL, 140      /THE MAX. NUMBER OF BCD SECONDS = 60.
        140      /THE MAX. NUMBER OF BCD MINUTES = 60.
        023      /THE MAX. NUMBER OF BCD HOURS = 13.
```

that the number of hours has been incremented from BCD 12 to BCD 13. If this is the case, the number of hours must be reinitialized to BCD 01. If the number of hours in the TOPVAL table (BCD 13) is equal to the content of HOURS, the JNZ to MIDNGT is not executed, instead, HOURS is reinitialized to BCD 01.

If the a.m./p.m. indicator must be changed (11:59:59 to 12:00:00), the 8080 jumps to CAMPM. Starting there, the address in register pair D is incremented so that it addresses the a.m./p.m. indicator. The A register is then loaded with the content of this memory location, the content of the A register is complemented, and the result is stored back in the same memory location. The first switch of the indicator will occur at noon. The next time that these instructions are executed, the current time will be 12:00:00 (midnight). This means that the indicator must be switched from p.m. to a.m., so the 377 (FF) stored in the a.m./p.m. memory locations is complemented to 0.

If desired, the a.m./p.m. indicator can also be initialized by the teletypewriter when the current time is entered (Example 3-5). To keep this software as simple as possible, register pair H would be loaded with the memory address for the a.m./p.m. indicator. The TTYIN subroutine would then be called so that the character A or P could be entered. If neither an A nor a P is entered, the 8080 would be programmed to ignore the character and then wait for either an A or a P. If an A is entered, the a.m./p.m. indicator would be set to 0. If a P is entered, the a.m./p.m. indicator is set to 377 (FF). This means that times would be entered as A10:56:03 or P02:13:23 (10:56:03 in the morning or 2:13:23 in the afternoon).

INTERRUPT-DRIVEN KEYBOARDS

In Chapter 7 of *8080/8085 Software Design—Book 1*^{2}, we described a number of different interfaces and subroutines that can be used with keyboards. In all of the software examples, however, the microcomputer was programmed to wait for a flag (see the ASCII keyboard examples) or to wait for a key closure to be sensed (see the 4×4 and 5×5 scanned keyboard examples). By using interrupts with these types of peripheral devices, we can eliminate all of the "flag/key sense" loops from the software.

In the previous chapter, an ASCII keyboard was interfaced to the microcomputer using a vectored interrupt (Fig. 2-3). A number of interrupt service subroutines for ASCII keyboards were also written and discussed (Examples 2-2 and 2-6). These examples should be reviewed if you are interested in interfacing an ASCII keyboard to the microcomputer using interrupts. Another type of keyboard that we discussed in *8080/8085 Software Design—Book 1*^{2} was a *scanned* keyboard. This type of keyboard does not have any flags or hardwired encoding logic associated with it. In fact, it is the responsibility of the 8080 to scan the keyboard to see if any keys are pressed. The 8080 also has to generate a unique code for each key. By using interrupts, the keyboard scan software can be simplified.

INTERRUPT-DRIVEN SCANNED KEYBOARDS

The interface for a 4×4 scanned keyboard is shown in Fig. 3-5. Part of this interface circuitry appears in Fig. 7-5 of *8080/8085 Software Design—Book 1*^{2}. However, the V, W, X, and Y input lines of the DM8095 are also wired to a four-input NAND gate (SN7420) and the output of this gate is gated through two D flip-flops, where the output of the second flip-flop goes to one of the priority-encoder interrupt interfaces shown in the the previous chapter. The NE555 oscillator has also been added to the interface. The addition of the flip-flops and the oscillator permit the interface to *debounce any* key closure. In *8080/8085 Software Design—Book 1*^{2}, the key closures were debounced using software. However, there is a specific reason why we chose to use hardware to debounce the key closures, and this point will be discussed in detail shortly.

Example 3-7 contains the interrupt service subroutine for the

Example 3-7: An Interrupt Service Subroutine for a 4X4 Scanned Keyboard

```
/THIS SECTION OF THE MAIN TASK "INITIALIZES"
/THE 4X4 MATRIX KEYBOARD.

         *000 000
START,   LXISP    /LOAD THE STACK POINTER WITH
         STACK    /A R/W MEMORY ADDRESS.
         0
         XRAA     /SET THE A REGISTER TO ZERO.
         OUT      /OUTPUT THIS TO THE KEYBOARD
         000      /(THE SN7475 LATCH).
         OUT      /CLEAR THE KEYBOARD'S
         200      /INTERRUPT FLAG.
         EI       /ENABLE THE INTERRUPT, AND
         •        /EXECUTE THE "MAIN TASK."
         •

/THIS INTERRUPT SERVICE SUBROUTINE SERVICES
/A MATRIX KEYBOARD THAT IS ARRANGED AS A 4X4
/MATRIX (4 ROWS OF 4 KEYS). THE CODE FOR THE KEY
/THAT IS PRESSED IS STORED IN R/W MEMORY.

         *000 050
KEYSCN,  PUSHD    /SAVE REGISTER PAIR D ON THE STACK.
         PUSHPSW  /THEN SAVE A AND THE FLAGS.
         JMP      /NOW CONTINUE THE KEY SCANNING
         SCN1     /AT A DIFFERENT SECTION OF
         0        /MEMORY.

         *030 162
SCN1,    LXID     /LOAD REGISTER PAIR D WITH THE
         376      /TEST WORD TO ACTIVATE ONE ROW OF KEYS
         003      /AND THE CODE FOR THE FIRST KEY.
NXTGRP,  MOVAE    /GET THE TEST WORD.
         OUT      /AND OUTPUT IT TO THE KEYBOARD.
         000
         RLC      /ROTATE THE TEST WORD LEFT ONE BIT.
         MOVEA    /AND THEN SAVE IT IN E.
         IN       /INPUT THE DATA FROM THE FOUR
         000      /ROWS OF KEYS.
         ANI      /SAVE ONLY THE FOUR LSB'S, WHICH CON-
         017      /TAIN THE ROW DATA.
         CPI      /SEE IF ANY KEYS ARE PRESSED BY
         017      /COMPARING 017 (0F) TO THE INPUT WORD.
         JNZ      /A KEY IS PRESSED IN THIS ROW, SO
         NXTKEY   /DETERMINE WHICH KEY IT IS.
         0
         DCRD     /NO KEYS ARE PRESSED IN THE TESTED ROW
         MOVAD    /SO DECREMENT THE KEY CODE BY ONE AND
         CPI      /SEE IF ALL FOUR ROWS HAVE BEEN TESTED
         377      /(377 = HEX FF).
         JNZ      /NOT ALL FOUR ROWS HAVE BEEN TESTED,
         NXTGRP   /SO TEST ANOTHER ROW.
         0
         HLT      /WHAT CAUSED THE INTERRUPT?
```

```
NXTKEY, RRC     /ROTATE THE ROW DATA INTO THE CARRY.
        JC      /THE CARRY IS A LOGIC ONE, SO
        UP4     /CALCULATE THE CODE FOR THE
        0       /NEXT KEY THAT CAN BE TESTED.
        MOVAD   /LOAD REGISTER A WITH THE KEY CODE.
        STA     /STORE THE KEY CODE IN R/W MEMORY.
        CHAR
        0
        POPPSW  /POP A AND THE FLAGS OFF THE STACK.
        POPD    /POP REGISTER PAIR D OFF THE STACK.
        OUT     /CLEAR THE KEYBOARD'S
        200     /INTERRUPT FLAG.
        EI      /ENABLE THE INTERRUPT AND
        RET     /RETURN.

UP4,    PUSHPSW /OTHERWISE, SAVE THE PSW ON THE STACK.
        MOVAD   /AND INCREASE THE KEY CODE IN D BY 4.
        ADI
        004
        MOVDA   /SAVE THE NEW KEY CODE IN D.
        POPPSW  /POP THE PSW OFF OF THE STACK.
        JMP     /AND THEN TRY FOR A ZERO CARRY
        NXTKEY  /AGAIN.
        0
```

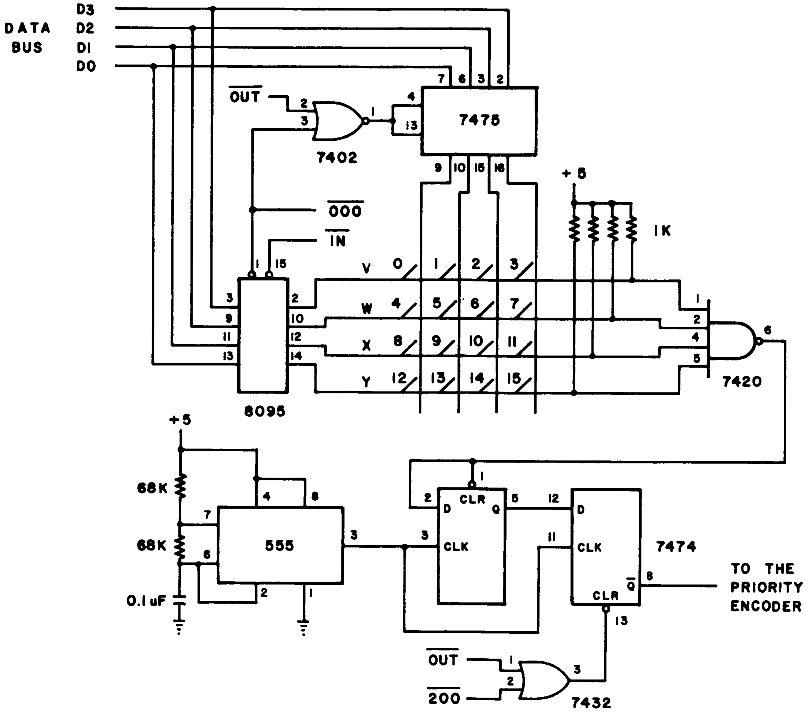

DATA BUS
D3
D2
D1
D0
OUT
7402
7475
000
IN
+5
1K
V
W
X
Y
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
8095
7420
+5
68K
68K
0.1 uF
555
CLR
D
Q
CLK
7474
TO THE PRIORITY ENCODER
OUT
200
7432

Fig. 3-5. A 4 × 4 matrix keyboard that is wired to the interrupt.

4×4 matrix keyboard. From this listing, you can see that the interrupt interface electronics must jam a RST5 instruction into the instruction register when the keyboard interrupts the microcomputer. When the keyboard does interrupt the 8080, register pair D and the PSW are saved on the stack. The remainder of the instructions in the interrupt service subroutine are stored in memory at SCN1.

This section of the subroutine is very similar to Example 7-13 in *8080/8085 Software Design—Book 1*[2]. The only differences are near NXTKEY. Suppose that the 8080 is interrupted by the keyboard, but when the interrupt service subroutine is executed, the 8080 cannot find any keys that are pressed. This should not happen, simply because most people cannot press and release a key in 10 or 20 μs. However, it may happen while you are in the process of debugging the interrupt service subroutine or the keyboard/interrupt interface hardware. If this situation does occur, the content of the D register will be decremented from 003 to 377 (03 to FF). This is an indication that no keys are pressed, so the 8080 halts just before NXTKEY.

Since a key can be pressed at any time, and an interrupt will occur when a key is pressed (unless the interrupt is disabled), we *cannot* return from this subroutine with the key code for the key that is pressed in the A register. Therefore, at NXTKEY, if a 0 is rotated into the carry (indicating that a key is pressed), the JC to UP4 is not executed. Instead, the key code in the D register is moved to the A register, and this value is stored in R/W memory. The PSW and register pair D are then popped off of the stack and the keyboard interrupt flag is cleared. The interrupt is then reenabled and the 8080 returns to the program that was interrupted. At some later time, the program that was interrupted can read the content of CHAR (the key code for the key that was pressed) to determine what actions should be performed.

As you can see, there are no instructions in Example 3-7 for debouncing the key when it is pressed, or later, when it is released. Instead, hardware is used in the interface so that all key closures are debounced. Remember, interrupts are used so that the 8080 can communicate with the peripheral device as quickly as possible. Once the 8080 has communicated with the peripheral, it can return to the task that was interrupted. Therefore, *we do not want to slow the microcomputer down by executing a 10 or 20 ms time delay subroutine so that the key closures are debounced with software.* For this reason, we included debounce hardware in the interface.

Another problem with using software to debounce the key closures is the fact that both the closing and opening of the switch must be debounced. It is very easy for the microcomputer to debounce the switch closure, simply because the keyboard is wired to the interrupt. This means that the 8080 can execute a 10 or 20 ms time delay subroutine after saving the registers on the stack and before the keyboard is "serviced." How will the 8080 detect when the key is released? *It will have to execute a loop in the interrupt service subroutine.* If the key is not released for two or three seconds, the 8080 will execute the "key sense" loop for two or three seconds while the key is pressed. When it is finally released, the time delay subroutine can be called so that the switch opening is debounced. The 8080 can then return to the task that was interrupted. Obviously, we do not want the 8080 to execute this type of loop in *any* type of interrupt service subroutine. Remember, interrupts are used when devices must be serviced as quickly as possible or at irregular intervals. Therefore, we do not want to tie the 8080 down in one subroutine. It is far better to let a small amount of interface hardware debounce all of the keyboard key closures and openings.

INTERRUPT-DRIVEN MULTIPLEXED LIGHT-EMITTING-DIODE DISPLAYS

In Chapter 7 of *8080/8085 Software Design—Book 1*[2] we also discussed multiplexed LED displays. A typical interface for a 10-

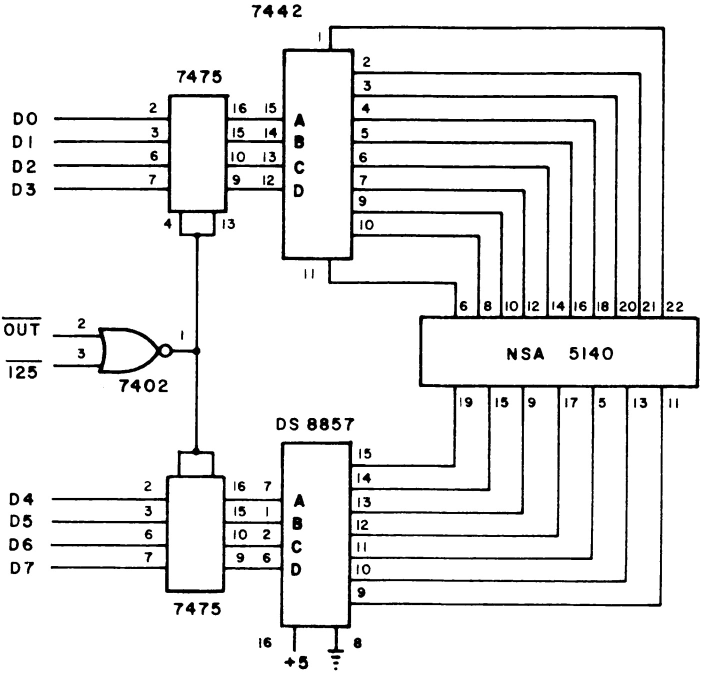

7442
7475
D0
D1
D2
D3
2
3
6
7
16 15
15 14
10 13
9 12
4
13
A
B
C
D
1
2
3
4
5
6
7
9
10
11
6 8 10 12 14 16 18 20 21 22
NSA 5140
19 15 9 17 5 13 11
OUT
125
2
3
1
7402
DS 8857
D4
D5
D6
D7
16 7
15 1
10 2
9 6
7475
15
14
13
12
11
10
9
16
8
+5

Fig. 3-6. A 10-digit multiplexed LED display.

Example 3-8: A Typical 10-Digit Multiplexed Display Program

```
/THIS PROGRAM DRIVES A 10-DIGIT, MULTIPLEXED,
/LIGHT-EMITTING DIODE (LED), SEVEN-SEGMENT DISPLAY.

DISPLA, LXIH    /LOAD REGISTER PAIR H WITH THE MEMORY
        120     /ADDRESS WHERE THE BCD DIGITS ARE STORED.
        004     /004 120 = HEX 0450.
        MVID    /LOAD D WITH THE FIRST DIGIT
        000     /POSITION THAT WILL BE ENABLED.
DISPL1, CALL    /DISPLAY THE FIRST TWO PACKED
        DIGIT   /BCD DIGITS.
        0
        INXH    /INCREMENT THE MEMORY ADDRESS.
        MOVAD   /GET THE DIGIT ENABLE WORD INTO A.
        CPI     /COMPARE IT TO THE
        012     /ELEVENTH DIGIT ENABLE CODE.
        JNZ     /HAVEN'T DISPLAYED ALL 10
        DISPL1  /DIGITS YET, SO DO TWO MORE.
        0
        JMP     /HAVE DISPLAYED ALL 10 DIGITS,
        DISPLA  /SO DISPLAY THEM ALL AGAIN.
        0
DIGIT,  MOVAM   /GET THE PACKED BCD WORD INTO A.
        RLC     /ROTATE THE FOUR LSB BITS INTO THE
        RLC     /FOUR MSB BITS.
        RLC
        RLC
        CALL    /THEN DISPLAY THIS DIGIT.
        OUTIT
        0
        MOVAM   /GET THE SAME WORD AGAIN.
OUTIT,  ANI     /SAVE ONLY THE FOUR MSB'S
        360     /(360 = HEX F0).
        ADDD    /ADD THE DIGIT ENABLE CODE.
        OUT     /OUTPUT THE EIGHT-BIT VALUE.
        125
        INRD    /INCREMENT THE DIGIT ENABLE CODE.
        RET
```

digit display is shown in Fig. 3-6. This is Fig. 7-10 in *8080/8085 Software Design—Book 1*[2]. A program that can be used to "drive" this display is listed in Example 3-8. This program does not use interrupts, it simply demonstrates how data values can be manipulated and output to the multiplexed display.

Unfortunately, this type of display needs "constant attention." This means that only one digit at a time can be turned on in the display, and each digit can only be on for a few milliseconds. Therefore, the microcomputer is constantly changing the digit that is turned on. Suppose that at some time, while the multiplexed display is being driven by the microcomputer, we have to execute a program that is very time sensitive; for example, a complex time-dependent control program or time delay subroutine. If this is the case, then one digit of the display cannot be left on for the 10 or 20 seconds while these instructions are being executed. The solution to this problem in *8080/8085 Software Design—Book 1*[2] was the addition of an integrated circuit to the interface electronics that provided an automatic power-off function for the display. The integrated circuit turned the display off if a new value was not output to the display every 10 ms. If a new value was not output to the display every 10 ms, the entire display was *blanked* (*no* digits were turned on). This hardware can be seen in Fig. 7-11 of *8080/8085 Software Design—Book 1*[2]. However, by using an interrupt, we can let the microcomputer do other tasks, while still having values displayed on the multiplexed display.

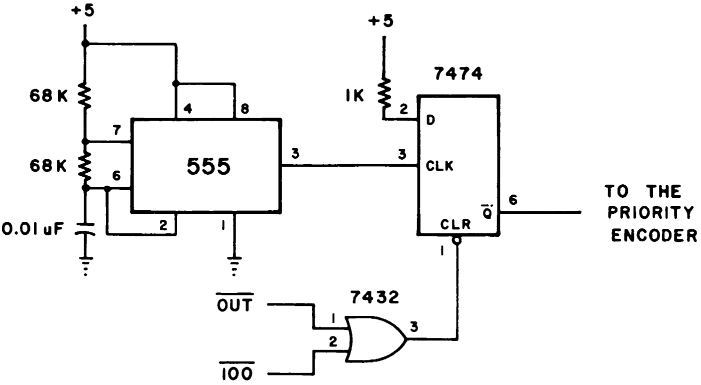


Fig. 3-7. A low-frequency oscillator that is wired to the interrupt.

What will cause the microcomputer to be interrupted? The multiplexed display does not have a flag that indicates that it needs servicing. This means that we will add a low-frequency oscillator to the multiplexed display interface (Fig. 3-7). The output of this oscillator will clock a D flip-flop, which is wired to a vectored priority interrupt interface. How often must the microcomputer be interrupted by this oscillator? For the display not to flicker, all 10 digits of the display must be turned on and off about 60 times a second. Since only one digit in the display can be turned on at a time, there will have to be 600 individual “digit times” in a second (10 digits × 60 times/second). Therefore, each digit must be turned on for 1.66 ms every 16.6 ms.

This means that the 8080 microcomputer could be interrupted 60 times a second, and each time it is interrupted, all 10 digits of information are displayed, each for 1.66 ms. Unfortunately, if this is done, the microcomputer will have very little time to perform other tasks. This will occur because the microcomputer will be inter-

Example 3-9: An Interrupt-Driven 10-Digit Multiplexed Display

```
        *000 000
START,  LXISP   /LOAD THE STACK POINTER BECAUSE SUB-
        STACK   /ROUTINES WILL BE CALLED AND INTER-
        0       /RUPTS CAN OCCUR.
        LXIH    /THEN LOAD REGISTER PAIR H WITH THE
        NMB     /MEMORY ADDRESS WHERE THE 10-DIGIT
        0       /NUMBER IS STORED.
        SHLD    /SAVE THIS VALUE IN R/W MEMORY
        ADDR    /SO THAT THE DISPLAY'S INTERRUPT
        0       /SERVICE SUBROUTINE CAN ACCESS IT.
        SHLD    /SAVE THE SAME ADDRESS IN ANOTHER TWO
        TEMPO   /R/W MEMORY LOCATIONS.
        0
        XRAA    /THEN SET A TO ZERO.
        STA     /AND SAVE THIS IN MEMORY AS
        DIGENB  /THE FIRST DIGIT ENABLE CODE.
        0
        OUT     /CLEAR THE OSCILLATOR'S
        100     /INTERRUPT FLAG.
        EI      /ENABLE THE INTERRUPT AND THEN
        •       /CONTINUE WITH THE REMAINDER
        •       /OF THE PROGRAM.
        •

/THIS IS THE INTERRUPT SERVICE SUBROUTINE FOR
/THE 10-DIGIT MULTIPLEXED DISPLAY.

        *000 050
DISPLA, PUSHH   /SAVE REGISTER PAIR H,
        PUSHD   /REGISTER PAIR D,
        PUSHPSW /AND THE PSW ON THE STACK.
        JMP     /THEN JUMP TO THE REMAINDER OF
        DISCNT  /THE INTERRUPT SERVICE SUB-
        0       /ROUTINE FOR THE DISPLAY.

        *060 131
DISCNT, LHLD    /LOAD REGISTER PAIR H WITH THE AD-
        ADDR    /DRESS WHERE THE 10-DIGIT NUMBER
        0       /TO BE DISPLAYED IS STORED.
        LDA     /LOAD THE A REGISTER WITH THE
        DIGENB  /CURRENT DIGIT ENABLE CODE TO
        0       /BE USED.
        MOVDA   /SAVE THE CODE IN REGISTER D.
DISPL1, MOVAM   /GET THE VALUE TO BE DISPLAYED.
        RLC     /ROTATE THE VALUE TO BE DISPLAYED
        RLC     /INTO THE 4 MSB'S OF THE A REGISTER.
        RLC
        RLC
        ANI     /SAVE ONLY THE FOUR MSB'S.
        360     /(360 = HEX F0).
        ADDD    /ADD THE CURRENT DIGIT ENABLE CODE.
        OUT     /OUTPUT THE RESULT TO THE
        125     /DISPLAY'S INTERFACE.
        MOVAD   /GET THE DIGIT ENABLE CODE IN A.
        INRA    /INCREMENT THE CODE.
        CPI     /HAS THE MOST SIGNIFICANT DIGIT
        012     /JUST BEEN DISPLAYED ?
        JNZ     /NO, THEN SAVE THE DIGIT ENABLE
        NOTYET  /CODE BACK IN MEMORY.
        0
        XRAA    /YES, THEN SET THE CODE TO ZERO.
        STA     /SAVE THE ZERO IN THE MEMORY LOCATION
        DIGENB  /USED FOR STORING THE CURRENT DIGIT
        0       /ENABLE CODE.
        LHLD    /THEN LOAD REGISTER PAIR H WITH THE
        TEMPO   /STARTING ADDRESS OF THE 10-DIGIT
        0       /NUMBER BEING DISPLAYED.
        JMP     /THEN SAVE THIS IN MEMORY, POP THE
        EXIT    /REGISTERS OFF OF THE STACK, ENABLE
        0       /THE INTERRUPT, AND RETURN.

NOTYET, STA     /SAVE THE CONTENT OF THE A REGISTER
        DIGENB  /(THE NEXT DIGIT ENABLE CODE
        0       /TO BE USED) IN R/W MEMORY.
        INXH    /INCREMENT THE MEMORY ADDRESS.
EXIT,   SHLD    /AND SAVE THE ADDRESS IN R/W
        ADDR    /MEMORY.
        0
        POPPSW  /POP THE PSW,
        POPD    /REGISTER PAIR D,
        POPH    /AND REGISTER PAIR H OFF OF THE STACK.
        OUT     /CLEAR THE OSCILLATOR'S
        100     /INTERRUPT FLAG.
        EI      /ENABLE THE INTERRUPT.
        RET     /AND RETURN.
```

rupted 60 times a second (once every 16.6 ms) and the interrupt service subroutine will require 16.6 ms (10 digits × 1.66 ms/digit) to be executed.

Another approach would be to interrupt the microcomputer 600 times every second (every 1.66 ms) and each time the microcomputer is interrupted, a value is displayed on a different digit in the display. It would appear that this method is just as poor as the previous method, simply because each digit must be on for 1.66 ms and the microcomputer is interrupted every 1.66 ms (600 times/second). However, we will use this latter method, because it is the best method of using an interrupt with a multiplexed display, as you will see (Example 3-9).

At the beginning of the program in Example 3-9, the stack pointer is loaded with a R/W memory address. Register pair H is then loaded with the memory address where the least-significant BCD digit of the 10-digit number that is to be displayed is stored. Since we want to display a 10-digit number, nine additional *consecutive* memory locations must be used to store the remaining BCD digits of the number. Each BCD digit must be stored in the four least-significant bits (D_0 through D_3) of each memory location. The 10-digit number is not stored in a packed BCD format.

Once register pair H is loaded with the address of the first digit, it is stored in R/W memory at ADDR and at TEMPO. Since we want to display the least-significant digit first, (however, it is just as easy to display the most-significant digit first), the 8080 sets the content of memory location DIGENB to 0. This will be used as the digit enable code of the display (for additional information on the operation of the display, refer to Chapter 7 of *8080/8085 Software Design—Book 1*[2]). Besides using 10 memory locations to store the number that is to be displayed, five additional memory locations are used by ADDR, TEMPO, and DIGENB. Once these five memory locations have been initialized, the oscillator interrupt flag is cleared and the interrupt is enabled. The 8080 can then execute the remaining instructions in the program.

Within 1.66 ms, the 8080 will be interrupted by the oscillator. When the 8080 is interrupted, it is vectored to 000 050 (0028). Starting at this address, the display is serviced. At DISPLA, register pairs H and D, along with the PSW, are saved on the stack. The 8080 then jumps to DISCNT.

At DISCNT, register pair H is loaded with the content of ADDR, which initially contains the address NMB, the memory address where the first digit to be displayed is stored. The A register is then loaded with the first digit enable code (initially 0), which is saved in the D register. This value will eventually be used to turn on, or enable, one digit in the display. At DISPL1, a BCD value is loaded into the A register from the memory location addressed by register pair H. This value is rotated to the left four times, into bits D_7 through D_4 of the A register. The ANI instruction sets bits D_3 through D_0 of the A register to 0. The digit enable code in the D register is then added to the BCD number contained in the A register.

The OUT instruction outputs the BCD value and the digit enable code to the display interface, so a new value is displayed on a different digit of the display. After the eight-bit value is output to the interface, the digit enable code is moved to the A register, where it is incremented. This generates the digit enable code for the next more-significant digit in the display. Since there are 10 digits in the display, the digit enable code will have a value between 000 and 011 (00 and 09). Therefore, the value 012 (decimal 10, hexadecimal 0A), is *not* a valid digit enable code. If this value is generated for the digit enable code, then the most-significant digit has just been displayed. Therefore, the least-significant digit must be the next digit displayed. After the digit enable code is incremented, the 8080 makes a decision.

If the digit enable code is not equal to 012 (0A), the JNZ to NOTYET is executed. At NOTYET, the incremented digit enable code is saved in R/W memory (DIGENB), and then the memory address in register pair H is incremented by 1. This new address, which points to the next BCD value in memory to be displayed, is then saved in ADDR when the instruction at EXIT is executed. The 8080 then pops the PSW and register pairs D and H off of the stack, and clears the oscillator interrupt flag. The interrupt is then reenabled and the 8080 returns to the task that was interrupted.

If the digit enable code is incremented to 012 (0A), the JNZ to NOTYET is not executed. Instead, the A register is set to 0 and this value is saved in DIGENB. The next time the display oscillator interrupts the 8080, this value will be used as the digit enable code. This is the code for the least-significant digit in the display. Register pair H is then loaded with the address where the least-significant data value is stored in memory, since it must be the next digit that is displayed. By jumping to EXIT, this address is saved in ADDR before the register pairs are popped off of the stack. After the register pairs are popped off of the stack, the interrupt flag is cleared, the interrupt is reenabled, and the 8080 returns to the task that was interrupted.

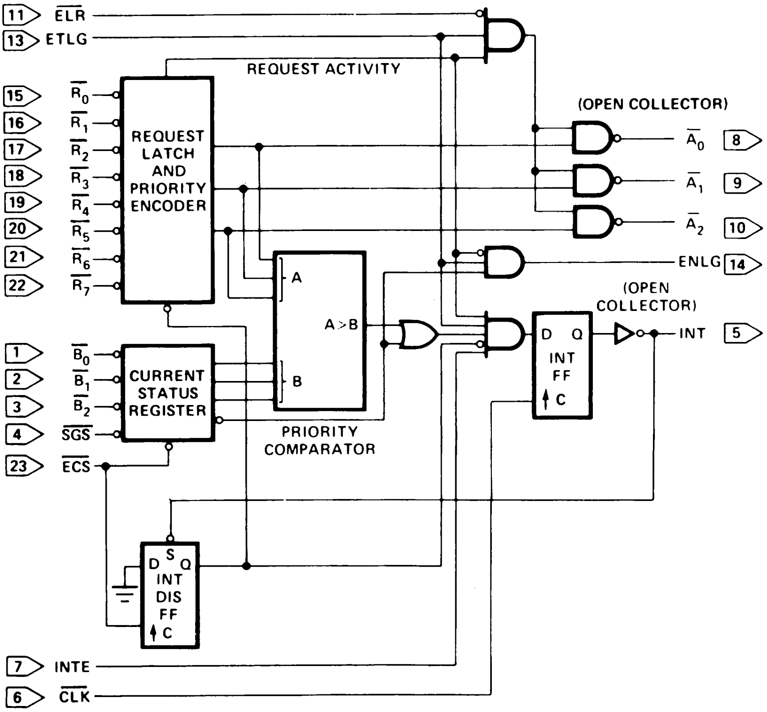
We have calculated that the 8080 needs a maximum of 131 μs to execute this interrupt service subroutine, even when the digit enable code is incremented to 012 (0A), and the digit enable code and the memory address ADDR have to be reinitialized.

If this is true, how is a single digit in the display turned on (enabled) for 1.66 ms? Because of the two four-bit latches (SN-7475) in the display interface (Fig. 3-6), the digit enable code and the data value are *latched* or *stored in the interface* until another OUT 125 instruction is executed. This will happen every 1.66 ms, because the OUT 125 instruction is only used in the interrupt service subroutine.

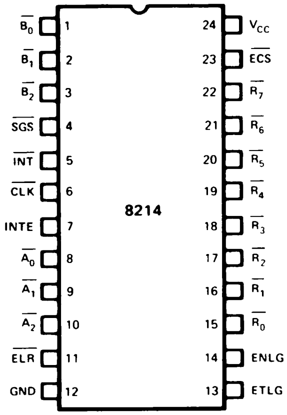
Since the 8080 needs only 131 μs to service the interrupt every 1.66 ms, the 8080 has a large amount of time to perform other tasks before another interrupt will occur. In fact, the 8080 spends only 7.8% of its time servicing the display. Of course, if this amount of time is too great, you simply have to program the 8080 with a DI (*D*isable *I*nterrupt) instruction before any time-sensitive tasks are performed. If these time-sensitive tasks involve servicing interrupt-driven floppy disks or cassettes, then the interrupt oscillator of the display will have to be disabled with interrupt hardware (Fig. 2-9) and interrupt masks. If the display is being used on an 8085-based microcomputer, then we could change the interrupt mask by executing a SIM instruction.

In the next section of this chapter, we will discuss the charac-

LOGIC DIAGRAM


11 $\overline{ELR}$
13 ETLG
REQUEST ACTIVITY
15 $\overline{R_0}$
16 $\overline{R_1}$
17 $\overline{R_2}$
18 $\overline{R_3}$
19 $\overline{R_4}$
20 $\overline{R_5}$
21 $\overline{R_6}$
22 $\overline{R_7}$
REQUEST LATCH AND PRIORITY ENCODER
(OPEN COLLECTOR)
$\overline{A_0}$ 8
$\overline{A_1}$ 9
$\overline{A_2}$ 10
ENLG 14
A
A > B
B
PRIORITY COMPARATOR
(OPEN COLLECTOR)
D Q
INT FF
C
INT 5
1 $\overline{B_0}$
2 $\overline{B_1}$
3 $\overline{B_2}$
4 $\overline{SGS}$
23 $\overline{ECS}$
CURRENT STATUS REGISTER
D S Q
INT DIS FF
C
7 INTE
6 $\overline{CLK}$

PIN CONFIGURATION


$\overline{B_0}$ 1
$\overline{B_1}$ 2
$\overline{B_2}$ 3
$\overline{SGS}$ 4
$\overline{INT}$ 5
$\overline{CLK}$ 6
INTE 7
$\overline{A_0}$ 8
$\overline{A_1}$ 9
$\overline{A_2}$ 10
$\overline{ELR}$ 11
GND 12
8214
24 V_{CC}
23 $\overline{ECS}$
22 $\overline{R_7}$
21 $\overline{R_6}$
20 $\overline{R_5}$
19 $\overline{R_4}$
18 $\overline{R_3}$
17 $\overline{R_2}$
16 $\overline{R_1}$
15 $\overline{R_0}$
14 ENLG
13 ETLG

PIN NAMES

INPUTS		
$\overline{R_0}$-$\overline{R_7}$	REQUEST LEVELS (R_7 HIGHEST PRIORITY)	
$\overline{B_0}$-$\overline{B_2}$	CURRENT STATUS	
$\overline{SGS}$	STATUS GROUP SELECT	
$\overline{ECS}$	ENABLE CURRENT STATUS	
INTE	INTERRUPT ENABLE	
$\overline{CLK}$	CLOCK (INT F-F)	
$\overline{ELR}$	ENABLE LEVEL READ	
ETLG	ENABLE THIS LEVEL GROUP	
OUTPUTS:		
$\overline{A_0}$-$\overline{A_2}$	REQUEST LEVELS	OPEN COLLECTOR
$\overline{INT}$	INTERRUPT (ACT. LOW)	
ENLG	ENABLE NEXT LEVEL GROUP	

Courtesy Intel Corp.

Fig. 3-8. The Intel 8214 priority interrupt control unit.

teristics of one of the interrupt controller integrated circuits that has been made specifically for the 8080, the 8214 priority interrupt control unit.

THE 8214 PRIORITY INTERRUPT CONTROL UNIT

The 8214 priority interrupt control unit (PICU) has some features that are very similar to those features available through the use of the circuit shown in Fig. 2-9. The pin configuration and logic diagrams for the 8214 PICU are shown in Fig. 3-8.

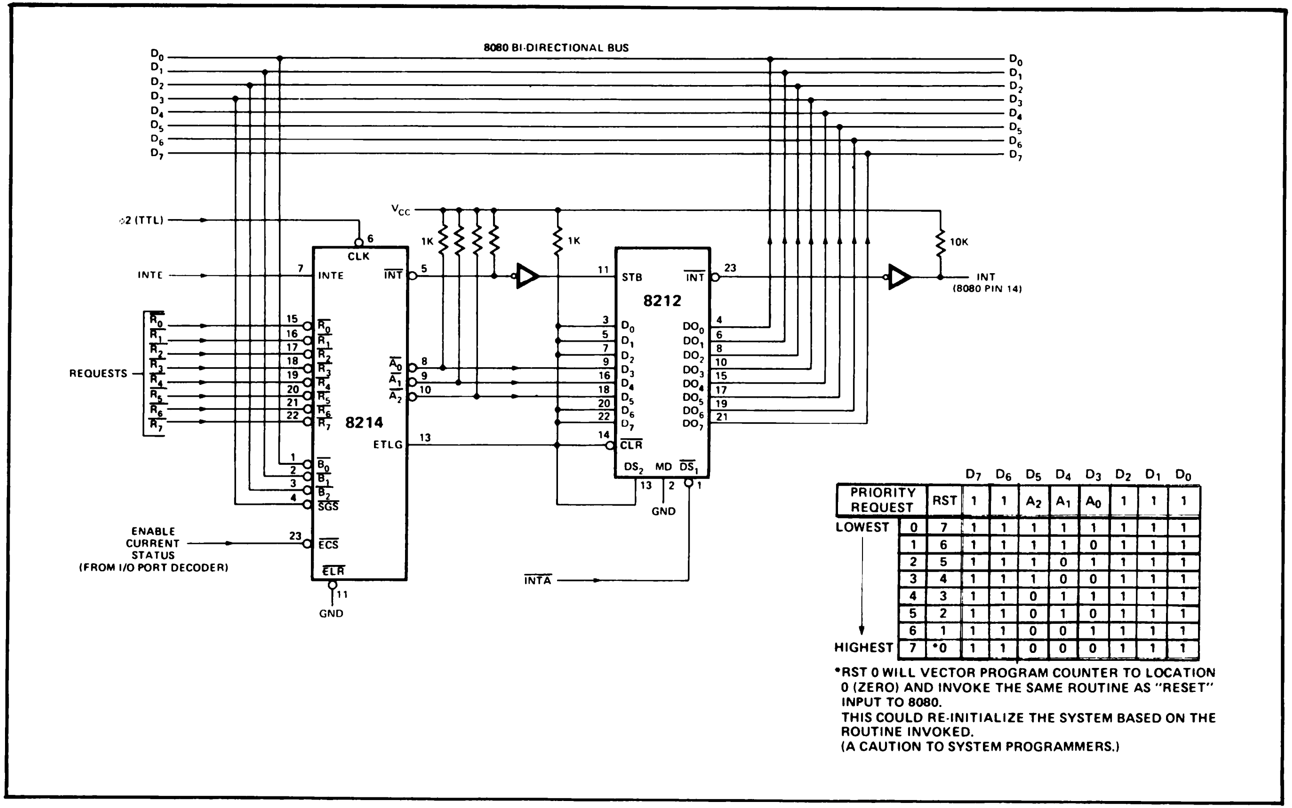
The 8214 PICU contains a priority encoder that has eight active-low inputs. $\overline{R_0}$ through $\overline{R_7}$. Input $\overline{R_7}$ has the highest priority of all eight inputs and $\overline{R_0}$ has the lowest priority. The code produced by the priority encoder is internally compared to the output of the *current status register*, which can be programmed by software. If the output of the encoder is greater than the output of the current status register, the interrupt output pin of the PICU, $\overline{\text{INT}}$, will go to a logic 0.

A number of 8214s can be cascaded together, so that the 8080 can be equipped with any number of priority encoded interrupts. For this reason, there are two cascade inputs (ETLG, $\overline{\text{ELR}}$) and one cascade output ($\overline{\text{ENLG}}$). For a system that needs only eight vectored priority interrupts, the ETLG pin (pin 13) is wired to +5 volts (V_{CC}, pin 24) and the $\overline{\text{ELR}}$ pin (pin 11) is grounded.

When the 8080 interrupt is enabled, an interrupt enable signal (INTE) is generated by the microprocessor integrated circuit (pin 16). This signal, and a clock, must be wired to the 8214. The clock can be any high-frequency clock, up to 12 MHz. If an 8224 clock generator is used with the 8080 microprocessor integrated circuit, the ϕ_2 (TTL) signal generated by the 8224 can be used as the clock for the 8214. For 8085 systems, the CLK OUT signal can be used to clock the 8214. As can be seen from the logic diagram (Fig. 3-8), the clock is used to gate the output of the five-input AND gate contained within the 8214 into the D flip-flop. The output of the flip-flop is used to indicate whether or not a device needs servicing.

There are only three outputs from the encoder section of the 8214. This means that additional interface circuitry is needed so that a different restart instruction is generated for each interrupting device. Fig. 3-9 shows one method that can be used to generate the restart instructions.

The encoded outputs of the 8214 ($\overline{A_0}$, $\overline{A_1}$, and $\overline{A_2}$) go to the three inputs of the 8112 (D_3 through D_5). The 8212 simply acts as an eight-bit interrupt instruction register. Because of the internal logic of the 8214, the $\overline{\text{INT}}$ output is active low for one clock cycle when


8080 BI-DIRECTIONAL BUS
D0
D1
D2
D3
D4
D5
D6
D7
φ2 (TTL)
INTE
REQUESTS
ENABLE CURRENT STATUS (FROM I/O PORT DECODER)
VCC
1K
1K
10K
INT (8080 PIN 14)
8214
CLK
INTE
INT
A0
A1
A2
ETLG
R0
R1
R2
R3
R4
R5
R6
R7
B0
B1
B2
SGS
ECS
ELR
GND
8212
STB
INT
DO0
DO1
DO2
DO3
DO4
DO5
DO6
DO7
CLR
DS2
MD
DS1
GND
INTA

		D7	D6	D5	D4	D3	D2	D1	D0
PRIORITY REQUEST	RST	1	1	A_2	A_1	A_0	1	1	1
LOWEST 0	7	1	1	1	1	1	1	1	1
1	6	1	1	1	1	0	1	1	1
2	5	1	1	1	0	1	1	1	1
3	4	1	1	1	0	0	1	1	1
4	3	1	1	0	1	1	1	1	1
5	2	1	1	0	1	0	1	1	1
6	1	1	1	0	0	1	1	1	1
HIGHEST 7	*0	1	1	0	0	0	1	1	1

*RST 0 WILL VECTOR PROGRAM COUNTER TO LOCATION 0 (ZERO) AND INVOKE THE SAME ROUTINE AS "RESET" INPUT TO 8080.
THIS COULD RE-INITIALIZE THE SYSTEM BASED ON THE ROUTINE INVOKED.
(A CAUTION TO SYSTEM PROGRAMMERS.)

Courtesy Intel Corp.

Fig. 3-9. A complete eight-level interrupt controller using the 8214.

one of the eight interrupt inputs is taken to a logic 0. Therefore, the $\overline{\text{INT}}$ output is inverted and wired to the strobe (STB) input of the 8212. This means that when one of the eight interrupt inputs of the 8214 is at a logic 0, the $\overline{\text{INT}}$ output of the 8212 will go to a logic 0. The 1000-ohm resistors are required on the four outputs ($\overline{\text{INT}}$ $\overline{\text{A}}_0$, $\overline{\text{A}}_1$, and $\overline{\text{A}}_2$) of the 8214 because they are *open collector* outputs.

Since the 8080 will only be interrupted if its interrupt pin (INT, pin 14) goes to a logic 1, the $\overline{\text{INT}}$ output of the 8212 has to be inverted. Note that the interrupt acknowlege signal ($\overline{\text{INTA}}$) is still used to gate the eight-bit restart instruction op code onto the data bus.

The current status register within the 8214 is used to determine which of the interrupt inputs ($\overline{\text{R}}_0$ through $\overline{\text{R}}_7$) will cause the 8080 to be interrupted. By writing a value out to the current status register, only the interrupts above the specified value will cause an interrupt to occur. For instance, interrupts $\overline{\text{R}}_4$, $\overline{\text{R}}_5$, $\overline{\text{R}}_6$ and $\overline{\text{R}}_7$ can be enabled by writing a 004 (04) out to the current status register. To enable only interrupt $\overline{\text{R}}_7$, a 001 (01) must be output to the 8214. Unfortunately, the 8214 cannot be programmed so that only interrupts $\overline{\text{R}}_0$, $\overline{\text{R}}_3$, $\overline{\text{R}}_4$, and $\overline{\text{R}}_7$ are enabled, and interrupts $\overline{\text{R}}_1$, $\overline{\text{R}}_2$, $\overline{\text{R}}_5$, and $\overline{\text{R}}_6$ are disabled. This is similar in function to the interrupt masks that were discussed in the previous chapter. However, for the sophisticated interrupt controller that we developed (Fig. 2-9), any combination of interrupts could be enabled and disabled. The three maskable interrupts that are built into the 8085 microprocessor integrated circuit (RST7.5, RST6.5, and RST5.5) can also be enabled and disabled in any combination.

One of the unusual features of the 8214 is the fact that if a 007 (07) is output to the current status register, all of the interrupts *except* $\overline{\text{R}}_0$ will be enabled. The $\overline{\text{SGS}}$ input of the 8214 can be used to solve this problem. If the $\overline{\text{SGS}}$ input is at a logic 1, all of the interrupt inputs ($\overline{\text{R}}_0$ through $\overline{\text{R}}_7$) will be enabled, regardless of the states of the $\overline{\text{B}}_0$, $\overline{\text{B}}_1$, and $\overline{\text{B}}_2$ inputs. If the $\overline{\text{SGS}}$ input is at a logic 0, then the $\overline{\text{B}}_0$, $\overline{\text{B}}_1$, and $\overline{\text{B}}_2$ inputs determine the "cutoff point" for the interrupt inputs.

To program the current status latch, the four inputs to it are connected to the data bus. By pulsing the *enable current status* pin ($\overline{\text{ECS}}$, pin 23), the content of the data bus can be gated into the current status register. The pulse used must be a combination of $\overline{\text{OUT}}$ ($\overline{\text{I/O W}}$) and a decoded device address, or $\overline{\text{MEMW}}$ and a decoded device address. As can be seen by the logic diagram in Fig. 3-8, the content of the current status register cannot be read back into the 8080.

The values that can be output to the current status latch, and

Table 3-4. Current Status Register Values for the 8214

$\overline{SGS}$	$\overline{B_2}$	$\overline{B_1}$	$\overline{B_0}$	Effect
0	0	0	0	No interrupt inputs are active.
0	0	0	1	$\overline{R_7}$ is active.
0	0	1	0	$\overline{R_7}$ and $\overline{R_6}$ are active.
0	0	1	1	$\overline{R_7}$ through $\overline{R_5}$ are active.
0	1	0	0	$\overline{R_7}$ through $\overline{R_4}$ are active.
0	1	0	1	$\overline{R_7}$ through $\overline{R_3}$ are active.
0	1	1	0	$\overline{R_7}$ through $\overline{R_2}$ are active.
0	1	1	1	$\overline{R_7}$ through $\overline{R_1}$ are active.
1	X	X	X	All interrupt inputs are active.

X = Don't care; a logic 1 or a logic 0.

the interrupts that they enable, are summarized in Table 3-4. The restart instructions that are generated by the interface shown in Fig. 3-8 are summarized in Table 3-5. The values listed in Table 3-5 are not what you would normally expect. In Chapter 2, the highest priority device always generated a RST7 instruction. Using the interface based on the 8214 and the 8212, the highest priority ($\overline{R_7}$) device generates a RST0 instruction.

Table 3-5. The Restart Instructions Generated by the Interrupt Hardware

Input Grounded	Priority	Restart Instruction
$\overline{R_7}$	Highest	RST0
$\overline{R_6}$	•	RST1
$\overline{R_5}$	•	RST2
$\overline{R_4}$	•	RST3
$\overline{R_3}$	•	RST4
$\overline{R_2}$	•	RST5
$\overline{R_1}$	•	RST6
$\overline{R_0}$	Lowest	RST7

As we have already discussed, a lower-priority device should not be able to interrupt the servicing of a higher-priority device, but a higher-priority device should be able to interrupt the servicing of a lower-priority device. Consider a microcomputer system that is interfaced to a floppy disk and a teletypewriter. The floppy disk should be able to interrupt the 8080 if it is servicing the teletypewriter, but the teletypewriter should not be able to interrupt the 8080 while it is servicing the floppy disk. To do this with the 8214, instructions can be executed in the interrupt service subroutine that reprogram the current status register.

If the floppy disk is wired to the $\overline{R_5}$ input, the teletypewriter would be wired to one of the inputs between $\overline{R_0}$ and $\overline{R_4}$. When the floppy disk interrupts the 8080, a 002 (02) would be output to the 8214 so that only devices with a higher priority ($\overline{R_6}$ or $\overline{R_7}$)

Example 3-10: Changing the Current Status Register Before Servicing Interrupt Devices

```
         *000 000
START,   MVIA      /LOAD THE CURRENT STATUS REGISTER
         010       /WITH 1000 (SGS = 1) SO THAT ANY
         OUT       /ONE OF THE EIGHT DEVICES CAN
         032       /INTERRUPT THE MICROCOMPUTER.
         EI        /ENABLE THE INTERRUPT.
         JMP       /THEN EXECUTE THE MAIN TASK.
         MT
         0

         *000 020
FLOPPY,  PUSHPSW /SAVE A AND THE FLAGS.
         MVIA      /THEN LOAD THE A REGISTER WITH A
         002       /NEW VALUE FOR THE CURRENT STATUS
         OUT       /REGISTER. OUTPUT IT TO THE 8214
         032       /(INTERRUPTS R6 AND R7 ENABLED).
         EI        /ENABLE THE INTERRUPT. ONLY DEVICES
         •         /R6 AND R7 CAN INTERRUPT THE MICRO-
         •         /COMPUTER WHILE IT IS SERVICING THE
         •         /DISK. WHEN DONE,
         MVIA      /ENABLE ALL EIGHT INTERRUPTS
         010       /BY CHANGING THE CURRENT STATUS
         OUT       /REGISTER (SGS = 1).
         032
         POPPSW    /RESTORE A AND THE FLAGS.
         EI        /ENABLE THE INTERRUPT.
         RET       /AND RETURN TO THE "MAIN TASK."

         *000 050
TTY,     PUSHPSW /SAVE THE PSW ON THE STACK.
         MVIA      /THEN LOAD THE A REGISTER WITH A
         005       /NEW VALUE FOR THE CURRENT STATUS
         OUT       /REGISTER. OUTPUT THIS VALUE SO
         032       /THAT R3 THROUGH R7 CAN STILL IN-
         EI        /TERRUPT THE MICROCOMPUTER.
         •         /THEN SERVICE THE TELETYPEWRITER.
         •
         •
         MVIA      /WHEN DONE, ENABLE ALL EIGHT INTERRUPTS
         010       /BY CHANGING THE CURRENT STATUS
         OUT       /REGISTER (SGS = 1).
         032
         POPPSW    /THEN POP THE PSW.
         EI        /ENABLE THE INTERRUPT.
         RET       /AND RETURN.
```

can interrupt the microcomputer. This will prevent the teletypewriter (device $\overline{R_2}$) from interrupting the 8080 while it is servicing the disk. When the 8080 services the teletypewriter (device $\overline{R_2}$), what value would be output to the 8214? The value 005 (05) would be output to the 8214 so that only devices with higher priority can interrupt the 8080 (see Table 3-6). Example 3-10 contains

Table 3-6. Values for the Current Status Register While the Interrupt Is Being Serviced

Interrupt	New Current Status Register Value			Enabled Interrupts
	Binary	Octal	Hex	
$\overline{R_7}$	00000000	000	00	None
$\overline{R_6}$	00000001	001	01	$\overline{R_7}$
$\overline{R_5}$	00000010	002	02	$\overline{R_7}$ and $\overline{R_6}$
$\overline{R_4}$	00000011	003	03	$\overline{R_7}$ through $\overline{R_5}$
$\overline{R_3}$	00000100	004	04	$\overline{R_7}$ through $\overline{R_4}$
$\overline{R_2}$	00000101	005	05	$\overline{R_7}$ through $\overline{R_3}$
$\overline{R_1}$	00000110	006	06	$\overline{R_7}$ through $\overline{R_2}$
$\overline{R_0}$	00000111	007	07	$\overline{R_7}$ through $\overline{R_1}$

the instructions that reprogram the 8214 when the teletypewriter and the floppy disk are serviced.

If interrupt $\overline{R_7}$ is being serviced, what device can interrupt the 8080? No device should be able to interrupt the 8080 while it is servicing the $\overline{R_7}$ device, because it has the highest priority. While this device is being serviced, we could output a 0 to the current status register in the 8214. However, this is not required. Why? When *any* device interrupts the 8080, the internal interrupt of the 8080 is automatically disabled. Therefore, the interrupt is not reenabled until the end of the $\overline{R_7}$ interrupt service subroutine. This will prevent any device from interrupting the 8080 while it is servicing this device.

A more recent interrupt controller integrated circuit, the 8259 programmable interrupt controller, is now available, and it can perform all of the functions of the 8214 and then some. However, because of the complexity of this device, it will not be discussed. It should be noted, however, that even though the 8214 is based on an older design, it probably can solve 95% of all the interrupt applications of the 8080 and 8085.

We have now discussed a number of interrupt applications. As you have seen, the interrupt service subroutines can be very simple or very complex. Unfortunately, most interrupt service subroutines are complex, because an interrupt can occur at *any time.* This means that if the interrupt service subroutine has not been properly written, or if there is an error in the interrupt hardware, the problem can be *very* difficult to locate. Because of this, it is best to avoid interrupts unless you absolutely have to use them. They do have their place, but all too often they are used "because they are there." Do not be fooled. A number of microprocessor and microcomputer manufacturers have interrupt controllers that can be used to connect up to 64 peripheral devices to the interrupt. If you have this many devices wired to the interrupt, you will spend a tremendous amount of time trying to get the entire system to work (both the hardware and the software). It would probably be easier to work with eight or 16 microcomputers, each of which would control eight or four interrupting devices.

REFERENCES

1. *Component Data Catalog, 1978.* Intel Corporation, Santa Clara, CA, 1978.
2. Titus, C. A., Rony, P. R., Larsen, D. G., and Titus, J. A. *8080/8085 Software Design—Book 1.* Howard W. Sams & Co., Inc., Indianapolis, IN, 1978.

4

Data Structures

In some of the previous chapters, we have not had large amounts of data that had to be accessed. We have not discussed searching the content of memory for particular values nor have we discussed sorting numeric values that are stored in memory. These topics will be discussed in the chapters that follow. Therefore, the different *structures* that can be used to store data in the microcomputer memory or in peripheral devices must be discussed.

Note that the title of this chapter is *Data Structures.* Knuth defines data as:

> data—(*Originally plural of the word "datum," but now used as singular or plural*): *Representation in a precise, formalized language, of some facts or concepts, often numeric or alphabetic values, in a manner which can be manipulated by a computational method.*[1]

We might obtain data from the result of some computation or it may have been read from a disk or an analog-to-digital converter. We might also have data values stored in memory that represent a person's name and age, using ASCII. Therefore, the data structures that we will be discussing can be used for numeric information or alphanumeric information.

Of course, the smallest unit of information that the 8080 (and all other eight-bit microprocessors) can operate on, or use, is a single *bit.* A bit can be represented by either a logic 1 or a logic 0. This single bit of information might tell us whether a door is open or closed or whether a tank is full of liquid or not. When eight bits of information are grouped together, we have an eight-bit *byte* or *word.* Since the 8080 memory is only eight bits "wide," we can store a single byte or word in each memory location. Of course, to different programs, a byte may represent different types of information. For instance, the byte 11000011 might represent the number of times a particular subroutine has been called, it might represent the ASCII character C, or it might indicate that four switches wired to some peripheral device are in the logic 0 state and four other switches are in the logic 1 state. By grouping or associating a number of bytes together, we may have a *table,* a *string,* an *array,* a *list,* or a *tree.*

Knuth states that, "Computer programs usually operate on tables of information. In most cases these tables are not simply amorphous masses of numerical values; they involve important *structural relationships* between the data elements."[1] The relationships between elements (entries) in a table might include one of the following: The element *n* is greater than the element *m* and less than the element *o.* If elements (entries) in the table are from an analog-to-digital converter, there is probably a time relationship between the various elements. This could mean that element *d* was read from the analog-to-digital converter some time before element *e* was read from the converter.

In addition, Knuth tells us that "The information in a table consists of a set of *nodes* (called 'records,' 'entities,' or 'beads' by some authors)."[1] We will often use the term *entry* rather than node, since it is slightly more descriptive in our examples. What is a node?

> **node**—*Each node consists of one or more consecutive words of the computer memory, divided into named parts called fields. In the simplest case, a node is just one word of memory, and it has just one field comprising the whole word.*[1]

If eight-bit data values are read from an analog-to-digital converter and are stored in memory in the form of a table, then each node might consist of a single data word. In this case, the field might be called ADC DATA or simply DATA.

Memory Location	Field
003 100	DATA } Node
031 250	DATA } Node
067 345	DATA } Node

If the time that the data was read from the analog-to-digital converter is also stored in memory, two memory locations might be used; one to store the data (DATA field), and the other to store the time (TIME field).

Memory Location	**Field**	
003 200	DATA	Node
003 201	TIME	
031 250	DATA	Node
031 251	TIME	
067 345	DATA	Node
067 346	TIME	

Of course, if the time is represented by a 16-bit number, the node would require three memory locations; one to store the data field, and two to store the time field. To distinguish one eight-bit byte of the time from the other, we might label one part of the time field TIME MSB and label the other part of the time field TIME LSB.

Memory Location	**Field**	
003 103	DATA	Node
003 104	TIME LSB	
003 105	TIME MSB	

or

Memory Location	**Field**	
003 103	DATA	Node
003 104	TIME MSB	
003 105	TIME LSB	

"The *address* of a node, also called a *link*, *pointer*, or *reference* to that node, is the memory location of its first word. The address is often taken relative to some 'base' location, ..."[1] However, in some cases, you will also know the absolute address of a particular node. For instance, the instruction,

```
LXIH
217
016
```

may load register pair H with the address of the fifth node (entry) in the table. Of course, if the table is moved about in memory, the address bytes (the second and third bytes) of the LXIH instruction would have to be changed.

LINEAR LISTS

"A *linear list* is a set of n nodes $X(1)$, $X(2)$, . . . $X(n)$ whose structural properties essentially involve only the linear (one-dimensional) relative positions of the nodes: the facts that if $n > 0$, $X(1)$ is the first node; when $1 < k < n$, the kth node $X(k)$ is preceded by $X(k-1)$; and $X(n)$ is the last node."[1]

SEQUENTIAL ALLOCATION

"The simplest and most natural way to keep a linear list inside a computer is to put the list items in sequential locations, one node after the other. We thus will have

$$\mathrm{LOC}(\mathrm{X}(j+1)) = \mathrm{LOC}(\mathrm{X}(j)) + c,$$

where c is the number of words per node. (Usually = 1. When $c > 1$, it is sometimes more convenient to split a single list into c 'parallel' lists, so that the kth word of node $\mathrm{X}(j)$ is stored a fixed distance from the location of the first word of $\mathrm{X}(j)$. We will continually assume, however, that adjacent groups of c words form a single node). In general,

$$\mathrm{LOC}(\mathrm{X}(j)) = \mathrm{L}_0 + cj,$$

where L_0 is a constant called a *base address*, the location of an artificially assumed node X(0)."[1]

LINKED ALLOCATION

"Instead of keeping a linear list in sequential memory locations, we can make use of a much more flexible scheme in which each node contains a link to the next node of the list (Fig. 4-1). Here, A, B, C, D, and E are arbitrary locations in the memory, and Λ is the null link. The program that uses this table in the case of sequential allocation would have to have an additional variable or constant whose value indicates that the table is five items in length, or else information would be specified by a 'sentinel' code within item 5 or in the following location. A program for linked allocation would have to have a link variable or constant that points to A, and from A all the other items in the list can be found."[1]

sentinel—*A special value placed in a table; e.g., to mark the boundaries of the table, designed to be easily recognizable by the accompanying program.*[1]

Sequential Allocation

Address	Contents
L + c:	Item 1
L + $2c$:	Item 2
L + $3c$:	Item 3
L + $4c$:	Item 4
L + $5c$:	Item 5

Linked Allocation

Address	Contents	
A:	Item 1	B
B:	Item 2	C
C:	Item 3	D
D:	Item 4	E
E:	Item 5	Λ

Fig. 4-1. Sequential and linked lists.

Sometimes, we will use the term "table terminator" rather than sentinel.

null link—*The null link is represented by some easily recognizable value that cannot be the address of a node.*[1]

CIRCULAR LISTS

"A slight change in the manner of linking furnishes us with an important alternative to the methods in the preceding section. A *circularly linked list* (briefly: a circular list) has the property that its last node links back to the first rather than to Λ. It is then possible to access all of the list starting at any given point; we also achieve an extra degree of symmetry, and if we choose we need not think of the list as having a 'last' or 'first' node."[1] An example of a circularly linked list is shown in Fig. 4-2.

Fig. 4-2. A circularly linked list.

Circularly Linked List

Address	Contents	
A:	Item 1	B
B:	Item 2	C
C:	Item 3	D
D:	Item 4	E
E:	Item 5	A

Although we will not discuss linked or circular lists, their descriptions were included as "food for thought." Those of you who are using linear lists that are stored in sequential locations may find that a linked or circular list is worth considering for some applications. There are numerous other data structures and the reader should refer to Knuth's work[1] for the last word on this subject.

Some of the other terms that you will find in the remainder of this book are defined as follows:

array—*A table which usually has an n-dimensional rectangular structure.*[1] A one-dimensional array is simply a linear list.

data structure—*A table of data including structural relationships.*[1]

sort—*The process of rearranging a given set of objects in a specific order.*[2]

string—*A finite sequence of zero or more symbols.*[1] See the Linear Lists section of this chapter.

symbol manipulation—*A general term for data processing, usually applied to nonnumeric processing such as manipulation of strings or algebraic formulas.*[1]

REFERENCES

1. Knuth, D. E. *The Art of Computer Programming. Volume 1, Fundamental Algorithms* (1968). *Volume 3, Sorting and Searching* (1973). Addison-Wesley Publishing Co., Reading, MA.
2. Wirth, N. *Algorithms + Data Structures = Programs.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

5

Searching

In some programming situations, we may have to search a list or table for a specific value or for the highest or lowest value. For example, suppose that tests are scored by computer. This means that someone will eventually have to write a program that will search for a particular student's name and then print out all of the grades associated with that name. In this example, a *node* (entry) in the table might consist of a NAME and GRADE *field.*

In another situation, a microcomputer might be used in an oil refinery. The microcomputer monitors the temperature of a cracking tower every 10 minutes, for 24 hours. At the end of the 24-hour period, the microcomputer has 144 temperatures stored in its memory. If the temperatures are stored in sequential memory locations, then it will be very easy for the microcomputer to determine the time at which a particular temperature was reached, or to determine the time when the temperature of the tower exceeded 300°C. We also might want to program the microcomputer so that it can find the lowest and highest temperatures that occurred. If this is done, we can relate these temperatures to any changes in the composition of the "cracked" petrochemical products from the tower. We also might like to know the number of times that the temperature went above or below a set temperature.

All of these problems can be solved by programming the 8080 microcomputer to search tables and lists. These lists might contain numeric, alphanumeric, or alphabetic information.

SINGLE-PRECISION LISTS (EIGHT BIT)

Finding the Smallest or Largest Unsigned Data Values in a List

One of the simplest subroutines that can be written can be used to find the smallest unsigned eight-bit number in a list. To keep the subroutine as general-purpose as possible, we should be able to specify any starting address for the list, and we should also be able to specify the number of *nodes* (entries or eight-bit numbers) in the list. The subroutine listed in Example 5-1 has these characteristics.

Example 5-1: Finding the Smallest Unsigned Eight-Bit Number in a List

```
/THIS SUBROUTINE WILL FIND THE SMALLEST UNSIGNED 8-BIT
/NUMBER IN A LIST. CALL THE SUBROUTINE WITH REGISTER
/PAIR H LOADED WITH THE LIST ADDRESS AND REGISTER PAIR
/D LOADED WITH THE NODE COUNT. THE SMALLEST NUMBER
/WILL BE IN THE B REGISTER WHEN THE 8080 RETURNS.

SMU8,    MOVBM   /LOAD THE B REGISTER WITH A NUMBER.
NEXTU8,  INXH    /INCREMENT THE LIST ADDRESS.
         DCXD    /DECREMENT THE NODE COUNT.
         MOVAD
         ORAE
         RZ      /RETURN IF THE COUNT IS ZERO.
         MOVAM   /OTHERWISE, GET A NUMBER INTO REGISTER A.
         CMPB    /COMPARE IT TO THE ONE IN REGISTER B.
         JC      /THE CONTENT OF THE A REGISTER IS LESS
         SMU8    /THAN THE CONTENT OF THE B REGISTER,
         0       /SO MOVE THE NUMBER FROM MEMORY TO B.
         JMP     /THE NUMBER IN THE B REGISTER IS THE
         NEXTU8  /SMALLER OF THE TWO, SO EXAMINE THE
         0       /NEXT NUMBER IN THE TABLE.
```

This subroutine (SMU8) must only be called with the starting address of the list in register pair H and the node count in register pair D. The first instruction in the subroutine loads the B register with the first node in the list. The list address is then incremented and the node count in register pair D is decremented. If the content of register pair D is decremented to 0, the 8080 will return from the subroutine with the smallest value that was found in the list in the B register. If the content of register pair D is decremented to a nonzero value, the 8080 moves the second node in the list to the A register. The first and second nodes are then compared. If the content of the A register is less than or equal to the content of the B register, the 8080 executes the JC instruction to SMU8. This means that the node in the list addressed by register pair H must be moved to the B register, since it is the smaller of the two. If the node in memory is greater than the node in the B register, the 8080 simply jumps back to NEXTU8, so that the next consecutive node in the list can be compared to the node in the B register.

What changes to the subroutine are necessary so that the 8080 finds the largest node in the list rather than the smallest? The simplest change would be to change the JC instruction to a JNC instruction. This change has been made to the subroutine listed in Example 5-2. This means that when the subroutine in Example 5-2 is called, the 8080 will only execute the JNC instruction if the content of the A register is equal to or greater than the content of the B register. This means that if the larger of the two nodes is contained in memory, the subroutine will move it from memory to the B register.

Example 5-2: Finding the Largest Unsigned Eight-Bit Number in a List

```
/THIS SUBROUTINE WILL FIND THE LARGEST UNSIGNED 8-BIT
/NUMBER IN A LIST. CALL THE SUBROUTINE WITH REGISTER
/PAIR H LOADED WITH THE LIST ADDRESS AND REGISTER PAIR
/D LOADED WITH THE NODE COUNT. THE LARGEST NUMBER
/WILL BE IN THE B REGISTER WHEN THE 8080 RETURNS.

LRGU8,   MOVBM   /MOVE A NUMBER TO THE B REGISTER.
NEXTU8,  INXH    /INCREMENT THE LIST ADDRESS.
         DCXD    /DECREMENT THE NODE COUNT.
         MOVAD
         ORAE
         RZ      /RETURN IF THE COUNT IS ZERO.
         MOVAM   /OTHERWISE, GET A NUMBER INTO REGISTER A.
         CMPB    /COMPARE IT TO THE ONE IN REGISTER B.
         JNC     /THE CONTENT OF THE A REGISTER IS MORE
         LRGU8   /THAN THE CONTENT OF THE B REGISTER,
         0       /SO MOVE THE NUMBER FROM MEMORY TO B.
         JMP     /THE NUMBER IN THE B REGISTER IS THE
         NEXTU8  /LARGER OF THE TWO, SO EXAMINE THE
         0       /NEXT NUMBER IN THE TABLE.
```

As in Example 5-1, the subroutine in Example 5-2 must be called with the starting address of the list in register pair H and the node count in register pair D. Of course, rather than have two almost identical subroutines, a single subroutine can be written that finds both the smallest and largest unsigned eight-bit nodes in a list.

Finding the Smallest and Largest Unsigned Eight-Bit Nodes in a List

The subroutine listed in Example 5-3 can be used to find the smallest and the largest unsigned eight-bit nodes in a list. Like the two previous subroutines, register pair H must be loaded with the starting address of the list, and register pair D must be loaded with the node count before the subroutine is called. When the 8080 returns from the subroutine, the smallest unsigned eight-bit node will be contained in the B register and the largest unsigned eight-bit node (number) will be contained in the C register.

In Example 5-3, the 8080 loads the first node (number) into both the C and B registers. The list address is then incremented

Example 5-3: Finding the Largest and the Smallest Unsigned Eight-Bit Numbers in a List

```
/THIS SUBROUTINE WILL FIND BOTH THE LARGEST AND SMALLEST
/EIGHT-BIT NUMBERS IN A LIST. CALL THE SUBROUTINE WITH
/THE LIST ADDRESS IN REGISTER PAIR H AND THE NODE COUNT
/IN REGISTER PAIR D. THE LARGEST NUMBER WILL BE IN THE
/C REGISTER AND THE SMALLEST IN THE B REGISTER WHEN THE
/8080 RETURNS.

SMLRU8, MOVCM   /MOVE A NUMBER INTO THE C REGISTER.
NEWSML, MOVBM   /MOVE THE SAME NUMBER INTO THE B REGISTER.
NEXTU8, INXH    /INCREMENT THE LIST ADDRESS.
        DCXD    /DECREMENT THE NODE COUNT.
        MOVAD
        ORAE
        RZ      /RETURN WHEN THE COUNT IS ZERO.
        MOVAM   /GET A NUMBER FROM THE LIST.
        CMPB    /COMPARE IT TO THE "SMALLEST" NUMBER.
        JC      /THE CONTENT OF MEMORY IS SMALLER THAN
        NEWSML  /THE CONTENT OF THE B REGISTER, SO MOVE
        0       /THE CONTENT OF MEMORY INTO THE B REGISTER.
        CMPC    /THE NUMBER IS GREATER THAN B, SO
        JC      /COMPARE IT TO THE "LARGEST" NUMBER IN MEMORY.
        NEXTU8  /THE CONTENT OF MEMORY IS SMALLER THAN
        0       /THE CONTENT OF C, SO GET THE NEXT NUMBER.
        MOVCM   /MOVE THE LARGER VALUE IN MEMORY INTO
        JMP     /THE C REGISTER AND THEN EXAMINE THE
        NEXTU8  /NEXT NUMBER IN THE LIST.
        0
```

and the node count is decremented. If the node count is equal to 0, the 8080 returns from the subroutine with the smallest unsigned eight-bit number in the B register and the largest unsigned eight-bit number in the C register. If the node count is decremented to a nonzero value, the 8080 moves the second node in the list to the A register and then compares the content of the B register to this value. If the content of the B register is larger than the content of the A register, the 8080 jumps to NEWSML, so that the smaller of the two numbers can be moved from memory to the B register.

If the node moved from memory to the A register is equal to or greater than the content of the B register, the 8080 does not execute the JC instruction. Instead, it compares the node in the C register to the content of the A register (from memory). If the content of the A register is less than the content of the C register, the 8080 jumps to NEXTU8, so that the next node in the list can be examined. If the content of the A register is equal to or larger than the content of the C register, the MOVCM instruction is executed, so that the "larger" of the two nodes is moved from memory to the C register. The 8080 then jumps to NEXTU8, so that the next node in the list can be examined.

As you can see from this example, it is almost as easy to find both the smallest and largest values, rather than just finding one or the other.

Finding the Smallest Signed (2s Complement) Eight-Bit Node in a List

Certainly in some applications we must use signed numbers. This may be as a result of the fact that an analog-to-digital converter that is interfaced to the microcomputer produces signed digital values, or we may be working with temperatures within the range of −100°C to +340°C. A summary of signed eight-bit numbers is shown in Table 5-1. As you see from this table, the range of signed numbers that can be represented by an eight-bit number is −128 to +127. We will consider −128 (octal 200, hex 80) to be the smallest signed eight-bit number, and +127 (octal 177, hex 7F) to be the largest signed eight-bit number.

Table 5-1. A Summary of Signed Eight-Bit Numbers

Signed Number	Binary	Octal	Hex
+ 127	01111111	177	7F
+ 126	01111110	176	7E
+ 125	01111101	175	7D
•	•	•	•
•	•	•	•
+ 2	00000010	002	02
+ 1	00000001	001	01
0	00000000	000	00
− 1	11111111	377	FF
− 2	11111110	376	FE
•	•	•	•
•	•	•	•
− 126	10000010	202	82
− 127	10000001	201	81
− 128	10000000	200	80

As was the case in the first section of this chapter, it is easiest to find either the smallest or largest node in a list. Therefore, the subroutine listed in Example 5-4 can be used to find only the smallest signed eight-bit number in a list.

In Example 5-4, we have assumed that register pair H is loaded with the starting address of the list and that register pair D is loaded with the node count *before* the subroutine is called. Therefore, at SMS8, the 8080 loads the first signed eight-bit number into the B register. The 8080 then increments the list address and decrements the node count. If the node count is now 0, the 8080 returns from the SMS8 subroutine with the smallest signed eight-bit number in the B register. If the node count is nonzero, the 8080 moves

Example 5-4: Finding the Smallest Signed (2s Complement) Eight-Bit Number in a List

```
/THIS SUBROUTINE WILL FIND THE SMALLEST 2'S COMPLEMENT
/(SIGNED) NUMBER IN A LIST. CALL IT WITH REGISTER PAIR
/H LOADED WITH THE ADDRESS OF THE LIST AND REGISTER PAIR
/D LOADED WITH THE NUMBER OF NODES IN THE LIST. THE
/SMALLEST NUMBER WILL BE IN THE B REGISTER WHEN THE
/8080 RETURNS.

SMS8,   MOVBM   /MOVE A NODE INTO THE B REGISTER.
NEXT8,  INXH    /INCREMENT THE MEMORY ADDRESS.
        DCXD    /DECREMENT THE NODE COUNT.
        MOVAD
        ORAE
        RZ      /RETURN IF THE COUNT IS ZERO.
        MOVAB   /OTHERWISE, GET THE ENTRY INTO A.
        XRAM    /EXCLUSIVE-OR IT WITH THE ENTRY IN MEMORY.
        JM      /IF THE SIGN BIT IS A ONE, ONE OF THE
        NEG     /TWO ENTRIES IS NEGATIVE. FIND OUT
        0       /WHICH ONE AND SAVE IT IN REGISTER B.
        MOVAB   /THE SIGNS OF THE NUMBERS ARE THE SAME.
        CMPM    /COMPARE THE TWO NUMBERS.
        JC      /(MEMORY) > B, SO EXAMINE THE
        NEXT8   /NEXT ENTRY (NODE) IN THE LIST.
        0
        JMP     /(MEMORY) < B, SO SAVE THE CONTENT
        SMS8    /OF MEMORY IN THE B REGISTER.
        0

NEG,    MVIA    /LOAD THE A REGISTER WITH
        200     /10000000.
        ANAM    /AND THE CONTENT OF A WITH MEMORY.
        JNZ     /A=200, SO THE CONTENT OF MEMORY
        SMS8    /IS NEGATIVE, SAVE IT IN THE B
        0       /REGISTER.
        JMP     /THE NEGATIVE NUMBER IS IN THE
        NEXT8   /B REGISTER ALREADY, SO EXAMINE
        0       /THE NEXT ENTRY (NODE).
```

the first node (in the B register) to the A register and exclusive-ORs it with the second node contained in the list. The 8080 executes these instructions to determine if one of the two nodes (numbers) is negative. If one of the numbers is negative (but not both), bit D_7 of the A register will be a logic 1 as a result of the XRAM instruction.

If one of the numbers is negative (bit D_7 of the A register is a logic 1), the 8080 executes the JM (*J*ump on *M*inus) to NEG. At NEG, the 8080 has to determine which of the two numbers is negative. If the number in the B register is negative, the 8080 can continue on with the list examination instructions. If the number contained in memory is negative, then it must be moved into the B register. Therefore, at NEG, the A register is loaded with 200 (binary 10000000, hex 80), and the content of memory addressed by register pair H is ANDed with it. If the node in memory is negative, the 8080 will execute the JNZ to SMS8, because bit D_7 of the A register is set to a logic 1 by the ANAM instruction. If the node in memory is positive, bit D_7 of the A register is cleared to a logic 0 and the JNZ to SMS8 is not executed. Instead, since the number in memory is positive, the 8080 simply jumps to NEXT8, because the negative number is already contained in the B register.

Note that the instructions at NEG are only executed when one of the signed numbers is positive and the other signed number is negative. This means that the negative number is the smaller of the two numbers. Therefore, the 8080 has to make sure that this number is eventually stored in the B register. If the numbers are both positive or negative, the instructions at NEG are not executed.

If both numbers are positive or negative, bit D_7 of the A register will be a logic 0 as a result of the XRAM instruction. Therefore, the JM to NEG is not executed. Instead, the 8080 moves the signed number contained in the B register to the A register and compares the content of memory addressed by register pair H to it. If the content of memory is larger than the node contained in the A register (and the B register), the 8080 jumps to NEXT8 so that the next consecutive node in the list can be examined. If the content of memory is less than or equal to the content of the A register, the 8080 executes the JMP to SMS8 so that this node is moved to the B register.

As you can see, the subroutine that finds the smallest *signed* eight-bit number is slightly more complex than the subroutine that finds the smallest *unsigned* eight-bit number (Example 5-1). It will also take the 8080 a little longer to examine a list that contains signed numbers, than it will to examine a list that contains unsigned numbers. In the next section of this chapter, we will discuss the software required to find the smallest and largest nodes in an unsigned 16-bit list and the smallest signed node in a 16-bit list.

DOUBLE-PRECISION LISTS (16 BIT)

Finding the Smallest and Largest Unsigned 16-Bit Nodes in a List

Rather than discuss a subroutine that finds *either* the smallest or the largest unsigned 16-bit numbers in a list, we will discuss a single subroutine that finds *both* the smallest and largest numbers. The subroutine that can do this is listed in Example 5-5.

When the subroutine listed in Example 5-5 is called, register pair H must contain the starting address of the list and register pair D must contain the node count. When the 8080 returns from the subroutine, the smallest unsigned 16-bit number will be con-

Example 5-5: Finding the Largest and the Smallest Unsigned 16-Bit Numbers in a List

```
/THIS SUBROUTINE WILL FIND THE SMALLEST AND LARGEST
/16-BIT UNSIGNED NUMBERS IN A LIST. CALL THE SUBROUTINE
/WITH THE LIST ADDRESS IN REGISTER PAIR H AND THE NODE
/COUNT IN REGISTER PAIR D. THE SMALLEST 16-BIT NUMBER
/WILL BE IN REGISTER PAIR B WHEN THE 8080 RETURNS AND
/THE LARGEST NUMBER WILL BE IN REGISTER PAIR D.

SAL16,   PUSHD   /SAVE THE NODE COUNT ON THE STACK.
         MOVEM   /GET AN MSBY INTO D AND THEN LOAD
         MOVCM   /THE SAME NUMBER INTO THE C REGISTER.
         INXH    /INCREMENT THE LIST ADDRESS.
         MOVDM   /GET AN MSBY INTO D AND THEN LOAD
         MOVBM   /THE SAME NUMBER INTO THE B REGISTER.
NEXT16,  INXH    /INCREMENT THE ADDRESS TO THE NEXT LSBY
         XTHL    /LIST ADDRESS ON THE STACK, NODE COUNT
         DCXH    /IN REGISTER PAIR H. DECREMENT
         MOVAH   /THE COUNT.
         ORAL
         XTHL    /LIST ADDRESS IS NOW IN REGISTER PAIR H.
         JNZ     /THE COUNT IS NONZERO, SO EXAMINE
         NOTYET  /ANOTHER NUMBER IN THE LIST.
         0
         POPH    /POP THE COUNT OFF OF THE STACK.
         RET     /AND THEN RETURN.
NOTYET,  MOVAE   /GET THE "LARGE" LSBY.
         SUBM    /SUBTRACT THE LSBY IN MEMORY.
         INXH    /INCREMENT THE MEMORY ADDRESS.
         MOVAD   /GET THE "LARGE" MSBY.
         SBBM    /SUBTRACT THE MSBY IN MEMORY.
         JC      /(MEMORY) > D AND E, SO MOVE THE
         LARGE   /16-BIT NUMBER IN MEMORY INTO
         0       /REGISTER PAIR D.
         DCXH    /DECREMENT THE ADDRESS TO THE LSBY.
         MOVAM   /GET AN LSBY FROM MEMORY.
         SUBC    /SUBTRACT THE LSBY OF "SMALL."
         INXH    /INCREMENT THE ADDRESS TO THE MSBY.
         MOVAM   /MOVE THE MSBY TO THE A REGISTER.
         SBBB    /SUBTRACT THE MSBY OF "SMALL."
         JNC     /(MEMORY) > OR = B AND C, SO SIMPLY
         NEXT16  /DECREMENT THE NODE COUNT, AND IF
         0       /REQUIRED, EXAMINE THE NEXT NODE.
SMALL,   DCXH    /DECREMENT THE MEMORY ADDRESS.
         MOVCM   /GET THE LSBY FROM MEMORY INTO C.
         INXH    /INCREMENT THE LIST ADDRESS.
         MOVBM   /GET THE MSBY INTO THE B REGISTER.
         JMP     /THEN SEE IF THE ENTIRE LIST
         NEXT16  /HAS BEEN EXAMINED.
         0

LARGE,   DCXH    /DECREMENT THE ADDRESS.
         MOVEM   /MOVE THE LSBY INTO THE E REGISTER.
         INXH    /INCREMENT THE LIST ADDRESS.
         MOVDM   /MOVE THE MSBY INTO THE D REGISTER.
         JMP     /THEN SEE IF THE ENTIRE LIST HAS
         NEXT16  /BEEN EXAMINED.
         0
```

tained in register pair B and the largest unsigned 16-bit number will be contained in register pair D.

When the 8080 executes the SAL16 subroutine, it first saves the node count (in register pair D) on the stack. The first node is then loaded into register pairs B and D. The LSBY of the node is loaded into the E and C registers and the MSBY of the node is loaded into the D and B registers. At NEXT16, the 8080 increments the address in register pair H so that it addresses the LSBY of the next node. The 8080 then exchanges the address in register pair H with the node count that is stored on the stack. After the node count is decremented, the 8080 moves the MSBY of the count to the A register and ORs it with the LSBY of the count. The flags are set or cleared by the ORAL instruction. The node count is then put back on the stack and the list address is put back into register pair H. Since XTHL instructions do not affect any of the flags, the 8080 will execute the JNZ to NOTYET if the node count (on the stack) is not equal to 0. If the node count is equal to 0, the 8080 pops the count off of the stack into register pair H and then returns from the subroutine. When the 8080 returns from the subroutine, the smallest unsigned 16-bit number will be in register pair B and the largest unsigned 16-bit number will be in register pair H.

If the node count is nonzero, the 8080 subtracts the node contained in memory from the "largest" node that has been found so far, which is contained in register pair D. Since the list contains unsigned 16-bit numbers, two eight-bit subtractions have to be performed in order to perform a 16-bit "comparison." If the node contained in memory is larger than the node contained in register pair D, the JC to LARGE is executed so that the node in memory is loaded into register pair D. The instructions at LARGE do just this. When these instructions have been executed, the 8080 jumps to NEXT16, so that the next 16-bit node in memory can be compared to the "largest" node found so far, and, if required, to the "smallest" node found so far.

If the node contained in memory is less than or equal to the node contained in register pair D, the JC to LARGE is not executed. Instead, the 8080 subtracts the "smallest" node found so far, which is stored in register pair B, from the node stored in memory. If the node stored in memory is equal to or greater than the node in register pair B, the 8080 executes the JNC to NEXT16, so that the next 16-bit number can be examined. If the node in memory is less than the node in register pair B, the JNC to NEXT16 is not executed. Instead, the 8080 moves the 16-bit unsigned number from memory to register pair B and then examines the next number in the list.

Finding the Smallest Signed (2s Complement) 16-Bit Node in a List

To find the smallest signed 16-bit number in a list, we have to perform many of the same operations that were performed to find the smallest signed eight-bit number in a list. However, because the 8080 cannot perform 16-bit comparisons, the 8080 must perform

Example 5-6: Finding the Smallest Signed (2s Complement) 16-Bit Number in a List

```
/THIS SUBROUTINE WILL FIND THE SMALLEST SIGNED (2'S COMPLEMENT)
/16-BIT NUMBER IN A LIST. CALL THE SUBROUTINE WITH THE
/LIST ADDRESS IN REGISTER PAIR H AND THE NODE COUNT IN REG-
/ISTER PAIR D. THE SMALLEST VALUE WILL BE IN REGISTER PAIR
/B WHEN THE 8080 RETURNS.

SMS16,  MOVCM   /GET THE LSBY INTO THE C REGISTER.
        INXH    /INCREMENT THE LIST ADDRESS.
        MOVBM   /GET THE MSBY (AND THE SIGN) INTO B.
NEXT16, INXH    /INCREMENT TO THE LSBY OF THE NEXT NODE.
        DCXD    /DECREMENT THE NODE COUNT.
        MOVAD
        ORAE
        RZ      /THE COUNT IS ZERO, SO RETURN.
        INXH    /INCREMENT TO THE MSBY.
        MOVAB   /GET THE OTHER MSBY.
        XRAM    /EXCLUSIVE-OR THE MSBY'S.
        JM      /IF THE SIGN BIT (D7) IS A ONE,
        NEG16   /ONE OF THE NUMBERS IS NEGATIVE, SO
        0       /FIND IT AND SAVE IT IN REGISTER PAIR B.
SAMES,  DCXH    /BOTH NUMBERS HAVE THE SAME SIGN.
        MOVAC   /GET ONE OF THE LSBY'S.
        SUBM    /SUBTRACT THE OTHER.
        INXH    /INCREMENT THE LIST ADDRESS.
        MOVAB   /GET ONE OF THE MSBY'S.
        SBBM    /SUBTRACT THE OTHER.
        JC      /(MEMORY) > REGISTER PAIR B,
        NEXT16  /SO EXAMINE THE NEXT NODE
        0       /IN THE LIST.
        DCXH    /DECREMENT H AND L TO THE LSBY.
        JMP     /(MEMORY) < REGISTER PAIR B, SO
        SMS16   /MOVE THE CONTENT OF MEMORY INTO
        0       /REGISTER PAIR B.

NEG16,  MVIA    /LOAD THE A REGISTER WITH
        200     /10000000.
        ANAM    /AND THE CONTENT OF A WITH MEMORY.
        JZ      /THE NUMBER IN MEMORY IS POSITIVE,
        NEXT16  /SO THE SMALLER NUMBER (THE NEGATIVE
        0       /NUMBER) IS ALREADY IN REGISTER PAIR B.
        DCXH    /DECREMENT THE ADDRESS TO THE LSBY.
        JMP     /THE NEGATIVE NUMBER IS IN MEMORY, SO
        SMS16   /MOVE IT TO REGISTER PAIR B.
        0
```

the comparison operation by subtraction. The subroutine that finds the smallest signed (2s complement) 16-bit number in a list is listed in Example 5-6.

When the 8080 calls the SMS16 subroutine, register pairs H and D must be loaded with the starting address of the list and the node count, respectively. The 8080 then loads the first signed 16-bit number into register pair B and decrements the node count in register pair D. If the node count is decremented to 0, the 8080 returns from the subroutine with the smallest signed 16-bit number in register pair B. The MSBY of the number will be in the B register and the LSBY of the number will be in the C register.

If the node count is decremented to a nonzero value, the 8080 increments the list address in register pair H so that it addresses the MSBY of the second node in the list. The signed MSBY of the number in register pair B is then moved to the A register, where it is exclusive-ORed with the signed MSBY of the number stored in memory. If one of the numbers is negative, bit D_7 of the A register will be set to a logic 1 by the XRAM instruction. If bit D_7 of the A register is a logic 1, the 8080 executes the JM to NEG16.

When the 8080 jumps to NEG16, it has to determine which of the two numbers is negative, and since the negative number is the smaller of the two numbers, the 8080 also has to ensure that this number is stored in register pair B. Therefore, at NEG16, the A register is loaded with binary 10000000 (octal 200, hex 80), and the signed MSBY of the number in memory is ANDed with it. If the number in memory is positive, the result of the ANAM instruction will be 0, so the JZ to NEXT16 is executed. This means that the negative number is already contained in register pair B, so the 8080 can examine the next number in the list. If the 8080 does not execute the JZ to NEXT16, the negative number is stored in memory, so the 8080 jumps to SMS16 so that the negative number in memory is moved to register pair B.

If the numbers are both positive or negative, a different sequence of instructions is executed after the XRAM instruction is executed. The JM to NEG16 is not executed, so the 8080 subtracts the number contained in memory from the number contained in register pair B. If the carry is set to a logic 1 as a result of this 16-bit "comparison," the content of register pair B is smaller than the content of memory. Therefore, the 8080 executes the JC to NEXT16 so that the next number in the list is examined. If the number in memory is smaller than or equal to the number in register pair B, the 8080 executes the jump to SMS16 so that the number in memory is moved to register pair B.

We have not included a subroutine in this chapter that can determine both the smallest and largest signed 16-bit numbers in

Example 5-7: Finding the Largest and Smallest Unsigned 24-Bit Numbers in a List

```
/THIS SUBROUTINE WILL FIND THE SMALLEST AND LARGEST 24-BIT
/UNSIGNED NUMBERS IN A LIST. CALL THE SUBROUTINE WITH
/THE LIST ADDRESS IN REGISTER PAIR H AND THE NODE COUNT IN
/REGISTER PAIR D. THE LARGEST AND SMALLEST NUMBERS WILL
/BE STORED IN R/W MEMORY WHEN THE 8080 RETURNS.

SLU24,   PUSHD    /SAVE THE NODE COUNT ON THE STACK.
         LXID     /LOAD REGISTER PAIR D WITH THE
         SMALL    /R/W MEMORY ADDRESS WHERE THE
         0        /SMALLEST 24-BIT NUMBER WILL BE STORED.
         LXIB     /LOAD REGISTER PAIR B WITH THE R/W
         LARGE    /MEMORY ADDRESS WHERE THE LARGEST
         0        /24-BIT NUMBER WILL BE STORED.
         MVIA     /LOAD THE A REGISTER WITH THE NUMBER OF
         003      /MEMORY LOCATIONS USED BY EACH NUMBER.
INITIL,  PUSHPSW  /SAVE THIS NUMBER ON THE STACK.
         MOVAM    /GET AN EIGHT-BIT BYTE FROM THE LIST.
         STAXD    /SAVE IT IN THE "SMALL" NUMBER.
         STAXB    /SAVE IT IN THE "LARGE" NUMBER.
         INXH     /INCREMENT THE LIST ADDRESS.
         INXD     /INCREMENT THE "SMALL" ADDRESS.
         INXB     /INCREMENT THE "LARGE" ADDRESS.
         POPPSW   /POP THE COUNT OFF OF THE STACK.
         DCRA     /DECREMENT THE COUNT.
         JNZ      /IF THE COUNT IS NONZERO, ANOTHER
         INITIL   /EIGHT-BIT BYTE MUST BE MOVED FROM THE
         0        /LIST TO "SMALL" AND "LARGE."
         MVIC     /LOAD THE C REGISTER WITH THE NUMBER
         003      /OF BYTES IN EACH NUMBER.
TEST24,  XTHL     /GET THE BYTE COUNT INTO H AND L.
         DCXH     /DECREMENT THE BYTE COUNT.
         MOVAH
         ORAL
         XTHL     /PUT THE BYTE COUNT BACK ON THE STACK.
         JNZ      /THE COUNT IS NONZERO, SO THERE IS
         TESTIT   /ANOTHER NUMBER IN THE LIST TO BE
         0        /EXAMINED.
         POPH     /THE COUNT IS ZERO, GET IT OFF THE STACK.
         RET      /AND RETURN.

TESTIT,  LXID     /H AND L POINT TO THE SECOND NUMBER IN THE
         SMALL    /LIST, SO LOAD REGISTER PAIR D WITH
         0        /THE ADDRESS FOR "SMALL."
         CALL     /THEN SUBTRACT THE 24-BIT NUMBER
         SUB3     /IN THE LIST FROM THE CONTENT
         0        /OF "SMALL."
         JNC      /THE CONTENT OF MEMORY IS SMALLER
         MOVE     /THAN THE CONTENT OF "SMALL,"
         0        /SO MOVE THE NUMBER FROM THE LIST TO "SMALL."
         LXID     /THE NUMBER WAS GREATER THAN "SMALL."
         LARGE    /IS IT LARGER THAN "LARGE."?
         0
         CALL
         SUB3
         0
         JNC      /NO, IT IS NOT LARGER THAN "LARGE," SO
         NOTLRG   /FIND THE NEXT NUMBER IN THE LIST.
         0        /AND DECREMENT THE NODE COUNT.
MOVE,    MOVBC    /MOVE THE COUNT FROM C TO B.
MOVE1,   MOVAM    /GET A BYTE FROM THE LIST.
         STAXD    /STORE IT IN "LARGE" OR "SMALL."
         INXH     /INCREMENT THE LIST ADDRESS.
         INXD     /INCREMENT THE STORAGE ADDRESS.
         DCRB     /DECREMENT THE BYTE COUNT.
         JNZ      /IF THE COUNT IS NONZERO,
         MOVE1    /ANOTHER BYTE MUST BE MOVED.
         0
         JMP      /MOVED ALL THREE BYTES, SEE IF THE
         TEST24   /NODE COUNT FOR THE LIST IS ALSO
         0        /ZERO.
NOTLRG,  INXH     /THE NUMBER IS LARGER THAN "SMALL" BUT
         INXH     /SMALLER THAN "LARGE," SO FIND THE NEXT
         INXH     /NUMBER IN THE LIST.
         JMP      /THEN SEE IF THE 8080 HAS
         TEST24   /REACHED THE END OF THE LIST.
         0
SUB3,    MOVBC    /LOAD THE BYTE COUNT INTO B.
         PUSHH    /SAVE THE ADDRESS OF THE NUMBER IN THE LIST.
         PUSHD    /SAVE THE ADDRESS OF "SMALL" OR "LARGE."
         XRAA     /SET A AND THE CARRY TO ZERO.
SUB3A,   LDAXD    /GET A BYTE FROM "SMALL" OR "LARGE."
         SBBM     /SUBTRACT A BYTE IN THE LIST.
         INXH     /INCREMENT THE LIST ADDRESS.
         INXD     /INCREMENT THE "SMALL" OR "LARGE"
         DCRB     /ADDRESS AND DECREMENT THE BYTE COUNT.
         JNZ      /IF ALL THREE BYTES HAVEN'T BEEN SUB-
         SUB3A    /TRACTED YET, DO ANOTHER.
         0
         POPD     /ALL THREE HAVE BEEN DONE, GET THE
         POPH     /ORIGINAL ADDRESSES OFF OF THE STACK.
         RET      /AND RETURN.

SMALL,   0        /THE SMALLEST 24-BIT NUMBER IS STORED
         0        /HERE.
         0
LARGE,   0        /AND THE LARGEST NUMBER IS
         0        /STORED HERE.
         0
```

a list. We leave this up to the reader. This subroutine should not be difficult to write, based on the previous examples.

TRIPLE-PRECISION LISTS (24 BIT)

The only example that we will discuss in this section of the chapter will determine the smallest and largest unsigned 24-bit numbers in a list. This subroutine is listed in Example 5-7. Because we would need six registers to store both the smallest and largest unsigned 24-bit numbers, the subroutine stores these two numbers in R/W memory. When this subroutine is called, the starting address of the list must be contained in register pair H and the node count must be contained in register pair D.

The first instruction in the SLU24 subroutine saves the node count on the stack. Register pair D is then loaded with the address at which the LSBY of the smallest number will be stored, and register pair B is loaded with the address at which the LSBY of the largest number will be stored. The A register is then loaded with 003 (03), because three memory locations are used to store each number in the list. This count is saved on the stack at INITIL. The 8080 then moves the LSBY of the first number in the list to the A register, and stores the content of the A register in the memory locations addressed by register pairs D and B. The memory addresses in register pairs H, D, and B are then incremented and the memory location counter is popped off of the stack into the A register. This value is then decremented and if the result is nonzero, the 8080 jumps to INITIL, where the decremented count is saved on the stack. The 8080 then proceeds to move two additional eight-bit bytes from the list to the memory locations reserved for the smallest unsigned 24-bit number and the largest unsigned 24-bit number. These instructions initialize SMALL and LARGE with the first number in the list.

Once the three bytes have been transferred, the C register is loaded with 003 (03). This is the number of eight-bit bytes in a 24-bit number. At TEST24, the 8080 decrements and tests the node count. If the node count is nonzero, the 8080 executes the instructions at TESTIT. If the node count is decremented to 0, the 8080 returns from the subroutine with the smallest unsigned 24-bit number stored in SMALL and the largest unsigned 24-bit number stored in LARGE.

At TESTIT, the 8080 loads register pair D with the address for the LSBY of the smallest number and then calls the SUB3 subroutine. This subroutine subtracts the content of three consecutive memory locations, starting at the memory location addressed by register pair H, from the content of the three memory locations addressed by register pair D. When the 8080 returns from this subroutine, the flags will reflect the result of this subtraction. Note that when this subroutine is called, register pair H addresses the LSBY of the first number in the list. *When the 8080 returns from the subroutine, register pairs D and H will contain the same addresses that they contained when the subroutine was called.* This means that it will appear that the SUB3 subroutine does not change the addresses contained in register pairs D and H.

When the 8080 returns from the subroutine, the JNC to MOVE will be executed if the 24-bit number contained in the list is smaller than or equal to the number contained in SMALL. The MOVE subroutine simply moves three bytes from the memory locations addressed by register pair H to three memory locations addressed by register pair D. After the 24-bit number is moved, the 8080 jumps to TEST24 so that the node count is decremented and tested. If the 24-bit number in the list is greater than the 24-bit number contained in SMALL, the 8080 loads register pair D with the address for LARGE and then calls the SUB3 subroutine. Remember, since the SUB3 subroutine does not change the address in register pairs D or H, register pair H still contains the address of the LSBY of the second number in the list.

When the 8080 returns from the SUB3 subroutine this time, the JNC to NOTLRG will be executed if the 24-bit number in the list is less than and not equal to the content of LARGE. If this is the case, the 8080 jumps to NOTLRG, where the memory address in register pair H is incremented three times, so that register pair H addresses the next 24-bit number (node) in the list. The 8080 then jumps to TEST24 where the node count is decremented and tested. If the 24-bit number in the list is greater than the content of LARGE, the JNC to NOTLRG is not executed. Instead, the 8080 executes the instructions at MOVE. These instructions cause the 24-bit number in the list to be moved to the memory locations used by LARGE.

COMMON SEARCH SUBROUTINE CHARACTERISTICS

All of the subroutines that we have discussed in this chapter so far have a number of common characteristics. *All of the subroutines require that the starting address of the list be loaded into register pair H and that the node count be loaded into register pair D.* This means that if you use an eight-bit search subroutine in a program and then have to upgrade the program to 16 or 24 bits, the subroutines will have the same starting address and node count requirements.

After much data processing, your program may have created lists that have to be searched. At some time, your program may create a list that has *no* nodes (entries). If this happens, and one of the search subroutines is called, the 8080 will compare the content of 65,536 nodes. The reason for this, of course, is that the node count is decremented from 000 000 (0000) to 377 377 (FFFF), so the 8080 will have to compare 65,536 nodes before returning from the subroutine.

You also must be careful in your interpretation of the results of a search if these subroutines are accidentally used to search lists that contain only one node (number or entry). This is particularly important when the subroutine returns both the smallest and largest nodes. If the list contains only one node, then the smallest and largest nodes found in the list will have equal values. This may or may not create problems for your data processing programs.

As you would expect, problems will also arise if you fail to load either register pair H or register pair D with the appropriate values. Suppose that you do not know the node count, but you do know the starting and final addresses of the list. Can the search subroutines still be used? Yes, the subroutines can still be used to find the smallest (and, if applicable, the largest) nodes. However, a new "front end" to the subroutine(s) will have to be written. A typical "front end" example is listed in Example 5-8. These instructions simply subtract the starting address from the final address and increment the difference by 1. This result is stored in register pair D while the starting address in register pair H remains unchanged. This assumes that the starting address is smaller than the final address.

Example 5-8: Calculating the Node Count From the Initial and Final Addresses of the List

```
/THIS MODIFICATION TO THE SEARCH SUBROUTINE
/CALCULATES THE DIFFERENCE BETWEEN THE STARTING
/AND FINAL ADDRESSES OF THE LIST.

ADRCAL, MOVAE   /GET THE LSBY OF THE FINAL ADDRESS.
        SUBL    /SUBTRACT THE LSBY OF THE INITIAL ADDRESS.
        MOVEA   /SAVE THE RESULT'S LSBY IN E.
        MOVAD   /GET THE MSBY OF THE FINAL ADDRESS.
        SBBH    /SUBTRACT THE MSBY OF THE INITIAL ADDRESS.
        MOVDA   /SAVE THE RESULT'S MSBY IN D.
        INXD    /INCREMENT THE DIFFERENCE BY ONE.
SMU8,   MOVBM   /LOAD THE B REGISTER WITH A NUMBER.
NEXTU8, INXH    /INCREMENT THE LIST ADDRESS.
        DCXD    /DECREMENT THE NODE COUNT.
        MOVAD
          •
          •
```

Note how these instructions have been added to the SMU8 subroutine (Example 5-1). *This sequence of instructions must only be used in front of subroutines that search single-precision (eight-bit) lists.* Why? Because in single-precision lists, each node (number) is stored in a single memory location. For a double-precision (16-bit) list, two memory locations are used to store a single node. Therefore, the number of nodes is equal to one-half the number of memory locations used by the list. For a triple-precision (24-bit) list, there are one-third the number of nodes as there are memory locations. This means that if a new front end is written for a search subroutine that examines double-precision (16-bit) lists, the content of register pair D must be divided by two after the difference is calculated and is incremented by 1.

How can the content of register pair D be *easily* divided by two? Simply rotate the content of the D register to the right using an RAR instruction, and then rotate the content of the E register to the right using an RAR instruction. *For this "division-by-two" to work, the content of the D register must be rotated to the right before the content of the E register.*

Although some readers might complain that we are filling pages with programs and subroutines that will not be used by many programmers, such as the 24-bit subroutine listed in Example 5-7, we do not feel this way. Our goal is to expose the reader and programmer to as many programming examples as possible. Even if the programs and subroutines are not used, there are many useful tricks, ideas, and comments that may eventually be used when you develop your own programs or modify someone else's.

SEARCHING ASCII STRINGS

Not only may the need arise to search lists for numeric information, you may also have to search a list for a specific sequence of ASCII characters (a *string;* see Chapter 4 on Data Structures). One of the most common uses for a program or subroutine that searches for ASCII strings is in the maintenance of mailing lists. There are many companies that sell and rent mailing lists, and they use computers to separate names by region, state, or zip code. Let us assume that you have a mailing list that is composed of six names and addresses and you want the 8080 to locate all of the names and addresses that have the same zip code. The names that you have on your list are as follows:

1. Apex Inc.
   20 Main Street
   Triangle, VA 24061

2. XYZ Blasting
   22 North Main
   Boom Boom, VA 24060

3. Easy Fire Insurance
   2280 W. Alpine
   Redhot, VA 24061

4. ABC Pencil Co.
   Arques Ave.
   Leadville, VA 24060

5. Clear Plastics, Inc.
   23 Hardy Ave.
   Cloudy, VA 24060

6. Ma's Fast Foods
   2105 NE West St.
   Speedy, VA 24061

Note: The numbers 1 through 6 are not stored in the list. They are for our reference only.

As you can see, these addresses have zip codes of either 24060 or 24061. As was the case with the multiple-precision-list search programs, there has to be some convention on how these names and addresses are stored in memory as ASCII characters. The simplest convention is to simply store all the ASCII values in consecutive memory locations. All the characters that are entered on the keyboard at some time or another are stored in memory, including the carriage return characters. At the end of the zip code for one address, the first character of the next name is stored. This can be seen in the following:

NAMES, •
•
•
A = 101
V = 126
E = 105
. = 056
CR = 015
C = 103
L = 114
O = 117
U = 125
D = 104
Y = 131
, = 054
SP = 040
V = 126
A = 101
SP = 040
2 = 062
4 = 064
0 = 060
6 = 066
0 = 060
CR = 015
M = 115
A = 101
’ = 047
S = 123
•
•

CR = Carriage return; SP = Space

At the end of the zip code for "Ma's Fast Foods," a 003 (03) will be stored as the *list terminator*. This will be used to notify the 8080 when it has reached the end of the list. The value 003 (03) is not a printing ASCII character, so it is a good choice as a *list terminator.* The terminator is required because it is not easy to determine, other than by counting the characters as they are entered into the microcomputer, how long the list is that contains the names and addresses. In all of the previous examples in this chapter, the number of nodes (numbers) in a list was known. However, in this case, if names and addresses are added and deleted from the list, it would not be easy to keep track of how many memory locations are actually used by the list. At this moment, do not be concerned with how we enter names and addresses into the microcomputer. Assume that the list of names and addresses is already stored in the microcomputer memory in the format that we just discussed.

Once the six names and addresses are stored in memory, you have to decide which of the two zip codes (24060 or 24061) that you want the 8080 to search for. If the 8080 searches for the zip code 24060 in all of the adddresses, the subroutine listed in Example 5-9 could be used.

This subroutine loads the starting address of the list into register pair D, so that it addresses the memory location that contains the ASCII value for the A in "Apex Inc." Register pair H is then loaded with the memory address where the first digit of the ASCII test zip code (in this case, 24060) is stored. This zip code is in the form of five ASCII characters, because the zip codes in the addresses are stored in the list in the form of ASCII characters. At CHECK, the first character in the list is moved to the A register and is compared to the value of 003 (03). Remember, this is just an arbitrary nonprinting character that is used to indicate to the 8080 when the end of the list is reached. The 003 (03) is stored in memory just after the zip code for "Ma's Fast Foods." If a 003 (03) is read from memory, the 8080 has searched the entire list, so it returns to the program that called START.

If the value read from memory is not 003 (03), it is compared to the first value in the test zip code, ASCII 2 (062, 32). If the two characters are equal in value, the 8080 executes the JZ to OK1. If the characters are not equal, the 8080 increments the memory address for the list and then attempts to match the next character in the list with 003 (03) and, if required, the first character in the test zip code. This means that the second character in the list is compared to the first character in the test zip code also.

When does the 8080 first execute the JZ to OK1? Refer to the initial listing of the names and addresses at the beginning of this section. The 8080 first jumps to OK1 when the 2 in the address "20 Main Street" is compared to the 2 in the zip code, 24060. When

Example 5-9: Searching a List of Names and Addresses for the Zip Code, 24060

```
/THIS SUBROUTINE WILL FIND THE FIVE DIGIT
/ZIP CODE IN THE ADDRESSES STORED IN THE LIST.

START,  LXID    /REGISTER PAIR D CONTAINS THE
        LIST    /INITIAL ADDRESS FOR THE LIST.
        0
AGAIN,  LXIH    /REGISTER PAIR H  CONTAINS THE
        ZIPCOD  /ADDRESS FOR THE FIRST DIGIT
        0       /OF THE TEST ZIP CODE.
CHECK,  LDAXD   /GET A LIST CHARACTER.
        CPI
        003     /AT THE END OF THE LIST YET?
        RZ      /YES, RETURN TO THE MAIN PROGRAM.
        CMPM    /NO, COMPARE IT TO THE 1ST DIGIT IN THE TEST ZIP CODE.
        JZ      /THERE IS A MATCH, SEE IF THERE ARE
        OK1     /FOUR MORE MATCHES.
        0
        INXD    /NO MATCH, INCREMENT THE LIST ADDRESS.
        JMP     /AND COMPARE IT TO THE FIRST DIGIT
        CHECK   /OF THE TEST ZIP CODE.
        0
OK1,    MVIB    /LOAD THE B REGISTER WITH THE NUMBER OF
        004     /ADDITIONAL MATCHES THAT MUST OCCUR.
OK2,    INXH    /INCREMENT THE TEST ZIP CODE LIST ADDRESS.
        INXD    /INCREMENT THE NAME/ADDRESS LIST.
        LDAXD   /GET THE NEXT ADDRESS CHARACTER.
        CMPM    /COMPARE IT TO THE TEST ZIP CODE.
        JNZ     /NOT A MATCH, TRY AGAIN
        AGAIN   /BY REINITIALIZING THE ZIP CODE
        0       /ADDRESS BUT NOT THE LIST ADDRESS.
        DCRB    /THERE WAS A MATCH, DECREMENT THE COUNT.
        JNZ     /FIVE MATCHES YET?
        OK2     /NO, TRY FOR ANOTHER MATCH.
        0
        JMP     /YES, A MATCH WAS FOUND.
        AGAIN   /TRY THE REMAINDER OF THE NAME
        0       /AND ADDRESS LIST.

ZIPCOD, 062     /THIS IS THE TEST ZIP CODE, 24060
        064     /ASCII 4
        060     /ASCII 0
        066     /ASCII 6
        060     /ASCII 0
```

the 8080 does jump to OK1, it will attempt to match the next four consecutive characters in the name and address list to the next four consecutive ASCII characters in the test zip code. Therefore, at OK1 the B register is loaded with 004 (04). The memory addresses in register pairs H and D are then incremented at OK2, and the 0 in “20 Main Street” is loaded from the list into the A register. This value is then compared to the ASCII 4 in the test zip code, 24060. Since these ASCII values are not equal, the 8080 jumps to AGAIN, where register pair H is loaded with the first memory address used by the test zip code. The address in register pair D is not changed, so that it still addresses a character in the list.

The second time that the 8080 executes the instructions at OK1 is when the 2 of the 24061 zip code is encountered in the first address. At OK1, the B register is loaded with the number of additional comparisons that should be made between the two zip codes, and at OK2, the 8080 continues to match the zip code characters. As expected, the 8080 matches the ASCII 4s that are contained in each zip code. Therefore, the 8080 does not execute the JNZ to AGAIN. Instead, the count in the B register is decremented from 004 to 003 (04 to 03). Since this result is nonzero, the 8080 jumps back to OK2 so that the next two characters can be compared. This loop will continue to be executed until either (1) a match does not occur between the test zip code and a character in the name/address list, or (2) the count in the B register is decremented to 0. In this example, the 8080 matches the 2406 in the list with the 2406 in the test zip code. However, the ASCII 1 in the name/address list is not equal to the ASCII 0 in the test zip code, so the JNZ to AGAIN is finally executed.

When will the content of the B register finally be decremented to 0? The first time that the B register is decremented to 0 will be when the zip code 24060 is found in the address for "XYZ Blasting." However, even though the 8080 successfully matches the two five-digit zip codes, the 8080 jumps back to AGAIN.

Does this program print the names and addresses when there is a match between the test zip code and one of the addresses in the list? The answer is no. No print instructions or subroutines are contained in the START subroutine.

Instead of jumping back to AGAIN, a series of instructions could be executed so that all of the names and addresses that contain the test zip code are printed. However, if a listing of these names and addresses is required, then there has to be some way of informing the 8080 where one address ends and another begins. For the names and addresses that are stored in the name/address list, the ASCII zip code characters are the last characters associated with a particular name before another name starts. This means that a name and address are "defined" as being stored in memory after a previous zip code and before another name. However, suppose the following name and address is added to the name/address list:

ABC Pencil Co.
Arques Ave.
Leadville, VA 24060
ATTN: Erasure Dept.

In this address, the zip code is not contained within the last five ASCII characters of the address. Instead, the ASCII string "Dept." is at the end of the address.

Of course, if names and addresses like this have to be added to the name/address list, the typists entering the names and addresses could be forced to always end the addresses with the zip code. However, this might badly scramble an address. Instead of doing this, an up-arrow (↑) could be stored after the last character in the address, no matter what it is. Just about any character could be used to *terminate* or *delimit* (end) the name and address, but characters that might appear in a name or address should not be used. Characters such as @, $, ¢, or * could be used. You would not use characters such as #, :, (,), or /, because these characters might be used in a name or an address. Now that a name terminator has been chosen, it must be added to the name/address list behind the last character of the address. The name/address list now appears as follows:

1. Apex Inc
20 Main Street
Triangle, VA 24061↑

2. XYZ Blasting
22 North Main
Boom Boom, VA 24060↑

3. Easy Fire Insurance
2280 W. Alpine
Redhot, VA 24061↑

4. ABC Pencil Co.
Arques Ave.
Leadville, VA 24060↑

5. Clear Plastics, Inc.
23 Hardy Ave.
Cloudy, VA 24060↑

6. Ma's Fast Foods
2105 NE West St.
Speedy, VA 24061↑

Note: The numbers 1 through 6 are not stored in the list. They are for our reference only.

Although adding the up-arrow (↑) was not necessary for the search subroutine to operate properly, it will be used by the section of the subroutine that prints the names and addresses on a teletypewriter or crt.

The search subroutine that was just developed (Example 5-9) can still be used to search the name/address list for matching zip codes. However, the JMP to AGAIN should be replaced by a JMP to BACKUP. The BACKUP (printer) sequence of instructions is listed in Example 5-10.

When the 8080 finishes a five-character match between the test zip code and an address in the list, the instructions at BACKUP are executed. At BACKUP, the 8080 looks for the up-arrow (↑) at the end of the *previous* address. The DCXD instruction decre-

Example 5-10: Printer Instructions for the Zip Code Search Subroutine

```
/THIS ADDITION TO THE PREVIOUS SUBROUTINE
/CAUSES THE 8080 TO BACK UP THE LIST ADDRESS
/UNTIL THE END OF THE PREVIOUS NAME IS FOUND.
/THEN THE NAME, ADDRESS, AND ZIP CODE ARE PRINTED.

        •
        •
BACKUP, DCXD    /THERE WAS A FIVE DIGIT MATCH.
        LDAXD   /BACK UP THE LIST ADDRESS TO THE
        CPI     /END OF THE PREVIOUS NAME.
        136
        JNZ     /FOUND THE ↑ AT THE END OF THE PREVIOUS
        BACKUP  /NAME YET? NO, KEEP BACKING UP.
        0
PRINT,  INXD    /FOUND THE ↑ , INCREMENT THE ADDRESS BY 1.
        LDAXD   /GET A CHARACTER FROM THE LIST.
        CPI     /IS IT A CARRIAGE RETURN?
        015
        JNZ     /NO, SEE IF IT IS THE ASCII
        NPCRLF  /CODE FOR AN UP-ARROW (↑).
        0
        CALL    /IT WAS A CARRIAGE RETURN, SO
        CRLF    /PRINT A CARRIAGE RETURN AND
        0       /LINE FEED.
        JMP     /THEN GET THE NEXT CHARACTER
        PRINT   /FROM THE LIST AND SEE WHAT
        0       /IT IS.
NPCRLF, CPI     /IS IT THE ↑ AT THE END OF THE NAME?
        136
        JZ      /YES, THEN DO NOT PRINT ANY MORE OF
        PCRLF   /THE CHARACTERS IN THE LIST. PRINT
        0       /A CR & LF THEN CONTINUE THE SEARCH.
        CALL    /THE CHARACTER WAS NOT THE ↑, SO PRINT
        TTYOUT  /IT.
        0
        JMP     /NOW GET ANOTHER CHARACTER AND CHECK IT.
        PRINT
        0
PCRLF,  CALL    /THE ↑ WAS FOUND, PRINT A CARRIAGE RETURN
        CRLF    /AFTER THE COMPLETE NAME AND ADDRESS.
        0
        JMP     /THEN SEARCH THE REMAINDER OF THE
        AGAIN   /LIST FOR ANY ADDITIONAL MATCHES.
        0
CRLF,   MVIA    /LOAD A WITH THE ASCII CODE
        015     /FOR A CARRRIAGE RETURN.
        CALL    /THEN PRINT IT ON THE TELETYPEWRITER
        TTYOUT  /OR CRT.
        0
        MVIA    /THEN LOAD A WITH THE ASCII CODE
        012     /FOR A LINE FEED.
        JMP     /THEN PRINT IT ON THE TELETYPEWRITER
        TTYOUT  /OR CRT.
        0
```

ments the list address and the LDAXD instruction reads one of the list characters into the A register. The CPI 136 instruction compares the ASCII value for the up-arrow character to the value read from the list. If the character that was read from memory is not an up-arrow, then the JNZ to BACKUP is executed. This causes the 8080 to examine the content of the memory location at the next lower memory address.

When the up-arrow at the end of the *previous* address is found, the 8080 will no longer execute the JNZ to BACKUP. Instead, the INXD instruction is executed at PRINT. Register pair D now addresses the first character in the name and address that contains the zip code that matched the test zip code. The first character in the name is then read from memory into the A register and is compared to the ASCII value for a carriage return. If an ASCII value for a carriage return is read from memory, the JNZ to NPCRLF is not executed. Instead, the 8080 calls the CRLF subroutine, which prints a carriage return and a line feed on the teletypewriter or crt. The 8080 then jumps back to PRINT, so that the character after the carriage return can be read from memory.

If the 8080 does not read the ASCII value for a carriage return from memory, the 8080 jumps to NPCRLF. Here the 8080 compares the ASCII value read from memory to the ASCII value for the up-arrow. If the up-arrow character was read from memory, the 8080 jumps to PCRLF. In this section of the program, the 8080 prints a carriage return and a line feed and then jumps to AGAIN. This means that the entire name and address that contained the matching zip code has been printed because the up-arrow at the end of the address was just read from memory. Therefore, the 8080 prints a carriage return and a line feed, and then continues on with the search for a matching zip code. Register pair D points to the first character in the next name, and register pair H is reinitialized with the address of the first character in the test zip code when the 8080 jumps to AGAIN.

If neither a carriage return nor an up-arrow is read from memory, the 8080 calls the TTYOUT subroutine, which prints the character on the teletypewriter or crt. By jumping back to PRINT, the 8080 can read the next consecutive character from the list and compare it to the ASCII values for the carriage-return character and the up-arrow character, and take the appropriate actions.

Why are the instructions listed in Example 5-11 required for the proper operation of the "printer" section of the subroutine? These instructions cause *both* a carriage return and a line feed to be printed on the teletypewriter or crt when the ASCII value for the carriage-return character is read from memory. As you can see from the organization of the name/address list, only the ASCII value for a carriage return is saved in the list. The ASCII value for a line feed is not saved in the list. Therefore, both a carriage return and a line feed must be printed when a carriage return is read from memory. One reason for not saving the ASCII value for the line-feed character is that if there are enough names in the list, we can save a little memory by just saving carriage-return characters.

Example 5-11: Instructions That Print a Carriage Return and a Line Feed

```
 •
 •
LDAXD     /GET A CHARACTER FROM THE LIST.
CPI       /IS IT A CARRIAGE RETURN?
015
JNZ       /NO, SEE IF IT IS THE ASCII
NPCRLF    /CODE FOR AN UP-ARROW (↑).
0
CALL      /IT WAS A CARRIAGE RETURN, SO
CRLF      /PRINT A CARRIAGE RETURN AND
0         /LINE FEED.
JMP       /THEN GET THE NEXT CHARACTER
PRINT     /FROM THE LIST AND SEE WHAT
0         /IT IS.
 •
 •
```

Have you located any *bugs* in the zip code search subroutine? The problem is associated with the first name in the list. What will happen if the first name and address contains the zip code that is being searched for? The search section of the subroutine will operate properly. However, when the instructions at BACKUP are executed, the 8080 will never find the up-arrow at the end of the previous address, because there is no previous address. This problem can be fixed by using one of two methods. The first method consists of having the typist enter an up-arrow into the microcomputer before any names or addresses are entered. Of course, the typist may forget to enter the up-arrow. The second method consists of writing a program that saves an up-arrow in memory automatically, before any names and addresses can be entered. These two methods are shown in Example 5-12.

In part A of Example 5-12, the typist must enter the up-arrow as the first character in the name/address list. In part B of Example 5-12, the up-arrow is automatically saved in the first memory location used by the name/address list. Note that only three *additional* memory locations are required so that the up-arrow is automatically saved at the beginning of the list. This is a small price to pay for the certainty that the up-arrow is stored at the beginning of the list.

Now that we have solved the problem of saving the up-arrow at the beginning of the list so that the search subroutine operates

Example 5-12: A Comparison of Two Different Up-Arrow Storage Techniques

(A) Using the Typist To Save the Up-Arrow

```
/IN THIS PROGRAM, THE TYPIST MUST SAVE THE
/UP-ARROW AS THE FIRST CHARACTER IN THE LIST.

ENTER,   LXISP    /SET THE STACK POINTER.
         350
         020      /HEX 10E8.
         LXIH     /REGISTER PAIR H POINTS TO THE
         LIST     /BEGINNING OF THE STORAGE AREA.
         0
NXTCHR,  CALL     /GET A TELETYPEWRITER CHARACTER.
         TTYIN
         0
         MOVMA    /SAVE IT IN MEMORY.
         INXH     /INCREMENT THE FILE ADDRESS.
         CPI      /WAS A CARRIAGE RETURN TYPED IN?
         015
          •
          •
```

(B) Having the Microcomputer Automatically Save the Up-Arrow

```
/IN THIS PROGRAM, THE UP-ARROW IS AUTOMATICALLY SAVED
/IN MEMORY AS THE FIRST CHARACTER IN THE LIST.

ENTER,   LXISP    /SET THE STACK POINTER.
         350
         020      /HEX 10E8.
         LXIH     /REGISTER PAIR H POINTS TO THE
         LIST     /BEGINNING OF THE STORAGE AREA.
         0
         MVIM     /SAVE AN ↑ AT THE BEGINNING OF
         136      /THE LIST.
         INXH     /INCREMENT THE LIST ADDRESS.
NXTCHR,  CALL     /GET A TELETYPEWRITER CHARACTER.
         TTYIN
         0
         MOVMA    /SAVE IT IN MEMORY.
         INXH     /INCREMENT THE LIST ADDRESS.
         CPI      /WAS A CARRIAGE RETURN TYPED IN?
         015
          •
          •
```

properly, we have the larger problem of writing a sequence of instructions that lets us save the names and addresses in a list. This sequence of instructions is listed in Example 5-13. The instructions listed in Example 5-13 enable you to enter a list of names and addresses into the microcomputer, and when the entire list has been entered, the microcomputer will search the list for a matching zip code. If any matches are found, the 8080 will print the names and addresses on the teletypewriter or crt. When the 8080 has searched the entire list, it will halt.

Example 5-13: A Program That Is Used To Input and Store Names and Addresses in a List

```
/THIS PROGRAM LETS YOU ENTER THE NAMES,
/ADDRESSES, AND ZIP CODES. IT THEN SEARCHES
/THE LIST FOR ANY MATCHES WITH THE TEST
/ZIP CODE. THE TEST ZIP CODE MUST ALREADY
/BE STORED IN MEMORY, AT THE SYMBOLIC
/ADDRESS "ZIPCOD."

ENTER,   LXISP   /SET THE STACK POINTER.
         350
         020     /HEX 10E8.
         LXIH    /REGISTER PAIR H POINTS TO THE
         LIST    /BEGINNING OF THE STORAGE AREA.
         0
         MVIM    /SAVE AN ↑ AT THE BEGINNING
         136     /OF THE LIST.
         INXH    /INCREMENT THE LIST ADDRESS.
NXTCHR,  CALL    /GET A TELETYPEWRITER CHARACTER.
         TTYIN
         0
         MOVMA   /SAVE IT IN MEMORY.
         INXH    /INCREMENT THE LIST ADDRESS.
         CPI     /WAS A CARRIAGE RETURN TYPED IN?
         015
         JNZ     /NO, NOT A CARRIAGE RETURN.
         NOTCR
         0
CRLFOK,  CALL    /YES, IT WAS A CARRIAGE RETURN, SO
         CRLF    /TYPE A CARRIAGE RETURN AND A LINE FEED.
         0
         JMP     /NOW GET ANOTHER CHARACTER.
         NXTCHR
         0
NOTCR,   CPI     /WAS A CTRL/C TYPED IN
         003     /TO TERMINATE THE INPUT ROUTINE?
         JNZ     /NOT A CTRL/C, SO GET ANOTHER
         NXTCHR  /CHARACTER FROM THE TELETYPEWRITER.
         0
SRCH,    LXID    /IT WAS A CTRL/C, D AND E POINT TO THE
         LIST    /LIST OF NAMES, ADDRESSES, AND ZIP CODES.
         0
START,   LXIH    /REGISTER PAIR H POINTS TO THE
         ZIPCOD  /TEST ZIP CODE STORED IN MEMORY.
         0
CHECK,   LDAXD   /GET A LIST CHARACTER.
         CPI     /IS IT THE CTRL/C STORED AT THE
         003     /END OF THE LIST?
         JNZ     /NO, SEE IF IT IS A ZIP CODE DIGIT.
         NOEND
         0
         HLT     /IT WAS THE CTRL/C, SO HALT.
NOEND,   CMPM    /COMPARE THE CHARACTER TO THE ZIP CODE.
         JZ      /THE FIRST DIGIT MATCHED, TRY
         OK1     /THE OTHER FOUR DIGITS FOR
```

```
        0       /A MATCH.
        INXD    /NO MATCH, INCREMENT THE LIST ADDRESS
        JMP     /AND TRY AGAIN.
        CHECK
        0
OK1,    MVIB    /LOAD THE B REGISTER WITH THE NUMBER OF
        004     /ADDITIONAL MATCHES THAT MUST OCCUR.
OK2,    INXH    /ONE DIGIT MATCHED, INCREMENT THE LIST
        INXD    /ADDRESS AND THE TEST ZIP CODE ADDRESS.
        LDAXD   /GET ANOTHER LIST CHARACTER.
        CMPM    /COMPARE IT TO THE TEST ZIP CODE.
        JNZ     /NO MATCH, CONTINUE EXAMINING
        START   /THE REMAINDER OF THE LIST.
        0
        DCRB    /THERE WAS A MATCH, DECREMENT THE COUNT.
        JNZ     /HAVE NOT CHECKED THE REMAINING
        OK2     /FOUR DIGITS YET, KEEP TRYING.
        0
BACKUP, DCXD    /THERE WAS A FIVE-DIGIT MATCH.
        LDAXD   /BACK UP THE LIST ADDRESS TO THE
        CPI     /END OF THE PREVIOUS NAME.
        136
        JNZ     /FOUND THE ↑ AT THE END OF THE PREVIOUS
        BACKUP  /NAME YET? NO, KEEP BACKING UP.
        0
PRINT,  INXD    /FOUND THE ↑, INCREMENT THE ADDRESS BY 1.
        LDAXD   /GET A CHARACTER FROM THE LIST.
        CPI     /IS IT A CARRIAGE RETURN?
        015
        JNZ     /NO, SEE IF IT IS THE ASCII
        NPCRLF  /CODE FOR AN UP-ARROW(↑)
        0
        CALL    /IT WAS A CARRIAGE RETURN, SO
        CRLF    /PRINT A CARRIAGE RETURN AND
        0       /LINE FEED.
        JMP     /THEN GET THE NEXT CHARACTER
        PRINT   /FROM THE LIST AND SEE WHAT
        0       /IT IS.
NPCRLF, CPI     /IS IT THE ↑ AT THE END OF THE NAME?
        136
        JZ      /YES, THEN DO NOT PRINT ANY MORE OF
        PCRLF   /THE CHARACTERS IN THE LIST. PRINT
        0       /A CR & LF, THEN CONTINUE THE SEARCH.
        CALL    /THE CHARACTER WAS NOT THE ↑ , SO PRINT
        TTYOUT  /IT.
        0
        JMP     /NOW GET ANOTHER CHARACTER AND CHECK IT.
        PRINT
        0
PCRLF,  CALL    /THE ↑ WAS FOUND, PRINT A CARRIAGE RETURN
        CRLF    /AFTER THE COMPLETE NAME AND ADDRESS.
        0
        JMP     /THEN SEARCH THE REMAINDER OF THE
        START   /LIST FOR ANY ADDITIONAL MATCHES.
        0
```

```
CRLF,   MVIA    /LOAD A WITH THE ASCII CODE
        015     /FOR A CARRIAGE RETURN.
        CALL    /THEN PRINT IT ON THE TELETYPEWRITER
        TTYOUT  /OR CRT.
        0
        MVIA    /THEN LOAD A WITH THE ASCII CODE
        012     /FOR A LINE FEED.
        JMP     /THEN PRINT IT ON THE TELETYPEWRITER
        TTYOUT  /OR CRT.
        0
```

At the beginning of Example 5-13, the 8080 loads the stack pointer with a R/W memory address, and then loads register pair H with the starting address of the name/address list. The MVIM instruction saves the ASCII up-arrow character in memory, and then the memory address in register pair H is incremented by 1. Starting at NXTCHR, the 8080 calls the TTYIN subroutine. When a key is pressed on the teletypewriter or crt, the 8080 will return with the seven-bit ASCII value for the key in the A register. This value is saved in memory and the memory address is incremented. If the key pressed was not the RETURN key, the 8080 jumps to NOTCR. If the RETURN key was pressed, the 8080 calls the CRLF subroutine so that a carriage return and a line feed are printed.

By jumping to NOTCR, the 8080 can determine if a CTRL/C was entered. If a CTRL/C was not entered, the 8080 jumps back to NXTCHR so that another ASCII character can be entered and saved in the list. If a CTRL/C was entered, the 8080 executes the instructions at SRCH. This means that the CTRL/C value is used to terminate the name and address entry section of the program. A CTRL/C should only be entered after *all* of the names and addresses have been entered. Note that the ASCII value for CTRL/C *is* saved in memory. This is the value that we used previously as the list terminator (Example 5-9). Of course, the CTRL/C "key" can only be pressed once.

When the CTRL/C is entered, the 8080 executes the instructions starting at SRCH, so that the names and addresses that contain the test zip code are printed on the teletypewriter or crt. When the 8080 is in the process of printing the zip codes, is the up-arrow also printed? No, the up-arrow is not printed. Instead, a carriage return and line feed are printed.

ENTERING A TEST ZIP CODE INTO MEMORY

How is a test zip code entered into memory? None of the program examples that we have discussed include any instructions that let us save the test zip code in memory. Instead, only the names and addresses for the name/address list can be entered. To enter the test zip code, a short addition can be made to the program listed in Example 5-13. This sequence of instructions is listed in Example 5-14. These instructions should be inserted into the program in front of the instruction at the symbolic address SRCH. This means that the names and addresses are entered into the list, then the test zip code is entered, and then the 8080 performs a search of the list.

After entering the names and addresses into the list, a CTRL/C is entered. This terminates the list with a 003 (03), and the 8080 begins to execute the instructions listed in Example 5-14. Register

Example 5-14: A Program To Enter the Test Zip Code

```
/THIS PROGRAM LETS YOU ENTER THE TEST ZIP
/CODE INTO THE 8080'S MEMORY. ANY CHAR-
/ACTERS ENTERED THAT ARE NOT VALID NUMBERS
/(0-9) WILL BE IGNORED.

          •
          •
        LXIH     /REGISTER PAIR H POINTS TO THE
        TEST     /STRING OF ASCII CHARACTERS.
        0
        CALL     /PRINT THE MESSAGE, WHICH SAYS,
        NXTLET   /"TEST ZIP CODE = "
        0
        LXIH     /NOW REGISTER PAIR H POINTS TO THE
        ZIPCOD   /SECTION OF R/W MEMORY WHERE THE
        0        /TEST ZIP CODE WILL BE STORED.
        MVIC     /C= THE NUMBER OF DIGITS TO ACCEPT.
        005
ZIPIN,  CALL     /GET A CHARACTER FROM THE TELETYPEWRITER.
        TTYIN
        0
        CPI      /IS IT LESS THAN ASCII 0?
        060
        JC       /YES, THEN IGNORE IT.
        ZIPIN
        0
        CPI
        072      /IS IT GREATER THAN AN ASCII 9?
        JNC      /YES, THEN IGNORE IT.
        ZIPIN
        0
        MOVMA    /IT WAS A 0 - 9, SAVE IT IN MEMORY.
        INXH     /INCREMENT THE MEMORY POINTER.
        DCRC     /DECREMENT THE DIGIT COUNTER.
        JNZ      /INPUT 5 DIGITS YET? NO, GET ANOTHER.
        ZIPIN
        0
          •      /YES, CONTINUE WITH THE REMAINDER
          •      /OF THE PROGRAM.
```

```
TEST,    215      /THIS IS THE ASCII MESSAGE STRING (CR)
         212      /LINE FEED
         324      /T
         305      /E
         323      /S
         324      /T
         240      /SPACE
         332      /Z
         311      /I
         320      /P
         240      /SPACE
         303      /C
         317      /O
         304      /D
         305      /E
         240      /SPACE
         275      /=
         240      /SPACE
         000      /MESSAGE TERMINATOR

NXTLET,  MOVAM    /GET A MESSAGE CHARACTER.
         CPI      /IS IT THE MESSAGE TERMINATOR?
         000
         RZ       /YES, THEN RETURN TO THE MAIN PROGRAM.
         CALL     /NO, PRINT THE CHARACTER.
         TTYOUT
         0
         INXH     /INCREMENT THE MEMORY ADDRESS.
         JMP
         NXTLET   /AND GET ANOTHER CHARACTER.
         0
```

pair H is loaded with the address of TEST, which is simply the starting address for a string of ASCII characters stored in memory. This string of characters is separate and distinct from the list that contains the names and addresses. After register pair H is loaded with this address, the NXTLET subroutine is called. At NXTLET, one of the ASCII characters is read from memory into the A register. This value is compared to 0. If a value of 0 was read from memory, the 8080 will return from the subroutine. If a nonzero value was read from memory, the TTYOUT subroutine is called so that the character is printed on the teletypewriter or crt. The memory address is then incremented and the 8080 jumps back to NXTLET so that another character can be read from memory.

Based on the program listing in Example 5-14, can you determine what the message is that is printed? The first characters that are printed are a carriage return and a line feed. The message, "TEST ZIP CODE = ", is then printed. Note that the printing stops when the 0 is read from memory at the end of this ASCII string.

After returning from NXTLET, register pair H is loaded with a R/W memory address where the test zip code is to be stored. The

C register is then loaded with the value 5, the number of digits contained in a zip code. This is also the number of times that the input loop, composed of the next nine instructions, is executed. After the C register is loaded, the TTYIN subroutine is called. The 8080 will return from this subroutine with the seven-bit ASCII value for the key that is pressed, in the A register. Since only the numbers 0 through 9 can be used to compose a zip code, the 8080 must ignore all nonnumeric characters. Any characters with ASCII values less than 060 or greater than 072 will be ignored.

Once a valid numeric key has been pressed (keys 0 through 9), the 8080 saves the value in memory by executing a MOVMA instruction. The memory address in register pair H is then incremented and the digit count contained in the C register is decremented. If the content of the C register is nonzero, the 8080 executes the JNZ to ZIPIN instruction. This loop permits the 8080 to accept and save in memory the successive digits of the zip code, until a five-digit code has been entered.

Once the test zip code has been saved in memory, the name/address list can be searched for any zip code matches. If a match occurs, the entire name and address, including the matching zip code, is printed. When the entire list has been searched, the 8080 will halt.

What will happen if the following keys are pressed while the 8080 is executing the instructions that permit the test zip code to be entered?

```
A12NEXT52+9
```

If these keys are pressed, the zip code 12529 will be stored in consecutive R/W memory locations as the test zip code.

TESTING THE ZIP CODE SEARCH PROGRAM

Using the names and addresses that were used as examples throughout this section of the chapter, we obtained the results listed in Example 5-15, using our 8080 microcomputer system. One limitation of this program is the amount of R/W memory that you have to have in your 8080 microcomputer. If it is assumed that each name and address requires 60 memory locations, then only 17 names and addresses can be stored in 1024 (1K) of R/W memory.

ONE FINAL PROGRAM BUG

There is one final bug in our zip code search program. Do you know what it is? Suppose the following name and address is contained in the name/address list:

Ed's Used Car Lot
24060 W. Main St.
San Diego, CA 93451↑

What will happen if the microcomputer operator wants to search all of the names and addresses in the list for the zip code 24060? Unfortunately, the address for "Ed's Used Car Lot" will also be printed, along with all of the names and addresses that contain the 24060 zip code. The problem is that the microcomputer cannot distinguish between a five-digit zip code and a five-digit street address or even a five-digit building number, lot number, employee number, or purchase order number, all of which may appear in a name and address.

What is the solution? There are a number of solutions to the problem. As one solution, another symbol (other than the up-arrow) could be stored just after the zip code. Of course, the typist would have to enter this symbol, along with the up-arrow. A name and address from the list might appear as follows:

Ed's Used Car Lot
24060 W. Main St.
San Diego, CA 93451$
ATTN: Foreign Car Sales↑

In this example, the dollar sign is used to indicate the end of the zip code. The only software modifications required would be in the zip code search section of the previous programs. After matching the five-digit test zip code with a zip code in the list, the program

Example 5-15: A Demonstration of the Zip Code Search Program

```
APEX INC.
20 MAIN ST.
TRIANGLE, VA 24061↑

XYZ BLASTING
22 NORTH MAIN
BOOM BOOM, VA 24060↑

EASY FIRE INSURANCE
2280 W. ALPINE
REDHOT, VA 24061↑

ABC PENCIL CO.
ARQUES AVE.
LEADVILLE, VA 24060↑

CLEAR PLASTICS, INC.
23 HARDY AVE.
CLOUDY, VA 24060↑
```

MA'S FAST FOODS
2105 NE WEST ST.
SPEEDY, VA 24061↑

TEST #1:

TEST ZIP CODE = 24060

XYZ BLASTING
22 NORTH MAIN
BOOM BOOM, VA 24060

ABC PENCIL CO.
ARQUES AVE.
LEADVILLE, VA 24060

CLEAR PLASTICS, INC
23 HARDY AVE.
CLOUDY, VA 24060

TEST #2:

TEST ZIP CODE = 24061

APEX INC.
20 MAIN ST.
TRIANGLE, VA 24061

EASY FIRE INSURANCE
2280 W. ALPINE
REDHOT, VA 24061

MA'S FAST FOODS
2105 NE WEST ST.
SPEEDY, VA 24061

OTHER TESTS:

TEST ZIP CODE = 23405

TEST ZIP CODE = 12358

Example 5-16: Searching for a Zip Code Delimiter

```
        •
        •
REALLY, INXD    /INCREMENT PAST THE FIFTH DIGIT.
        LDAXD   /GET THE CHARACTER INTO A.
        CPI     /IS IT THE DOLLAR SIGN?
        077     /(077=HEX 3F).
        JNZ     /NOT THE DOLLAR SIGN, THEREFORE
        AGAIN   /A FIVE-DIGIT MATCH DID NOT OCCUR
        0       /BETWEEN TWO ZIP CODES, KEEP LOOKING.
BACKUP, DCXD    /THERE WAS A FIVE-DIGIT MATCH.
        •
        •
```

would then check to see if the dollar sign is stored in the list just after the last digit of the zip code. This software is listed in Example 5-16.

These instructions must be inserted just before BACKUP. Only when a five-digit match occurs are these instructions executed. The program simply increments the address contained in register pair D so that it addresses the character after the fifth digit in the matching number. If the matching number was a zip code in an address, then register pair D should address the memory location that contains the dollar sign stored after the zip code. The LDAXD loads this character into the A register, where it is compared to the immediate data byte, 077. If the ASCII value read from memory is not that of a dollar sign, the JNZ to AGAIN is executed. This jump will be executed if a street address, or any other number in the address not terminated with a dollar sign matches the test zip code. If a zip code in the address matches the test zip code, the JNZ to AGAIN is not executed. Instead, the 8080 executes the instructions at BACKUP. Of course, for this to work properly, a dollar sign must be saved (entered) after *every* zip code.

There are a number of other methods that can be used to identify the zip code in the address. Can you think of any? Some of the solutions may be based on the structure of the list. For instance, isn't a carriage return stored just before every street address and just after every zip code? What are the chances of a zip code appearing in the second line of a name and address? Doesn't an abbreviation for a state (NY, CA, VA, MI) always precede the zip code? The answers to these questions should indicate to you other methods that can be used to distinguish a zip code from a street address or other five-digit number.

One final note: If a dollar sign is stored after every zip code, the instructions listed in Example 5-17 should be added to your pro-

Example 5-17: Preventing the Dollar Sign From Being Printed

```
        •
        •
PRINT,  INXD    /FOUND THE ↑, INCREMENT THE ADDRESS BY 1.
        LDAXD   /GET A CHARACTER FROM THE LIST.
        CPI     /IS IT THE DOLLAR SIGN STORED
        077     /AFTER THE ZIP CODE?
        JZ      /YES, THEN DO NOT PRINT IT.
        PRINT   /SO GET ANOTHER CHARACTER FROM THE LIST.
        0
        CPI     /NOT THE DOLLAR SIGN, IS IT A
        015     /CARRIAGE RETURN?
        •
        •
```

gram. This sequence of instructions prevents the dollar sign from being printed if the address contains a matching zip code.

REFERENCE

1. Sternheim, E. "Scanning Bidirectional Lists With One Link." *EDN,* January 20, 1978, pp 37-38.

6

Sorting

Sorting is defined as the rearrangement of data values in a list, table, record, or file in a descending, ascending, or alphabetical order. Why do programmers sort data values? Quite often, a program will operate much faster if the data values that it must access are sorted. Suppose that you have two magnetic tapes. One tape contains all of the names of students that received an "F" in a course during the last semester. The other tape contains all of the class schedules for all of the students for the next semester. If a student received an "F" in a course, the student will have to take the course over again during the next semester. Therefore, the courses that were failed will have to be included in the class schedules of the students.

If the two magnetic tapes are not sorted alphabetically by name, it will take the microcomputer a *long* time to find the "Fs" for one student on one tape and then find the student's class schedule on the other tape. If the names are sorted alphabetically on both tapes, it will take the microcomputer far less time to find a student who failed a course and then find the appropriate class schedule on the other tape.

In many cases, one microcomputer will not be interfaced to two magnetic tape drives. However, a microcomputer might be designed into an *intelligent* magnetic tape drive controller. The host computer might order the microcomputer in the controller to sort the tenth record or file on the magnetic tape. If there is no microcomputer in the tape drive, the host computer will have to do the sorting.

Table 6-1 contains a number of sorted and unsorted lists. In this table, there is one list that is sorted in an ascending order (the third list) and one list that is sorted in a descending order (the

Table 6-1. Sorted and Unsorted Lists

Memory Location	Sorted	Unsorted	Sorted	Unsorted
X	5	3	1	4
X+1	4	2	2	2
X+2	3	4	3	3
X+3	2	1	4	5
X+4	1	5	5	1

first list). If the data values in a list are not organized in an ascending, descending, or alphabetical order, then the list is not sorted (the second and fourth lists in Table 6-1).

There are a number of different sorting techniques that can be used to sort data values. These techniques include *straight insertion sort, binary insertion sort, Shell's sort, bubble sort, quick sort, heap sort, merge-exchange sort,* and *two-way merge sort.* In general, all of these sorting techniques can be classified as one of the following five techniques:[1]

1. Insertion Method
2. Exchange Method
3. Selection Method
4. Merge Method
5. Distribution Method

The two methods of sorting that we will discuss are the insertion and exchange methods. For details about the other sorting methods, Knuth's book[1] is excellent.

SORTING NUMERIC VALUES

Insertion Sorting (Straight Insertion Sort)

The simplest insertion sorting method, and the one that we will discuss, is the *straight insertion* method. Assume that the following numbers are stored in consecutive R/W memory locations:

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	5	3	4	2	1

The 5 is stored in the memory location with the lowest memory address, and the 1 is stored in the memory location with the highest address. If the straight insertion method of sorting is used, the 8080 would compare the 3 stored in X+1 to the 5 stored in X. Since the 3 is less than the 5, the 5 would be *moved up* one memory location, from X to X+1, and the 3 would be written into the memory location (X) that *originally* contained the 5. The order of the list is now

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	3	5	4	2	1

The next data value in the next higher memory location, the 4 stored in X+2, is then compared to the 5 stored in X+1. Since the 4 is less than the 5, but the 4 is more than the 3, the 5 is *moved up* one memory location to X+2, and then the 8080 writes the 4 into memory location X+1, the memory location that contained the 5.

The 2, stored in X+3, is the next node that must be examined by the 8080. Since the 2 is less than the 3, 4, and 5, all three of these data values are moved up one memory location. The 5 is moved up to X+3, the 4 to X+2, and the 3 to X+1. The 2 is then written into memory location X. The list now appears as

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	2	3	4	5	1

When the 8080 examines the last node in the list, the 1, it determines that it is less than all the other nodes in the list. Therefore, all of the other nodes are moved up one memory location, and the 1 is written into memory location X.

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	1	2	3	4	5

Since the 8080 has examined, and moved if required, the last node in the list, the list is now sorted. Only when all of the nodes in the list have been examined can this conclusion be reached. As you can see from the list example that we just used, the straight insertion method of sorting requires that the 8080 make a number of comparisons. Based on these comparisons, data values may be moved to different places within the list. The subroutine listed in Example 6-1 uses the straight insertion method of sorting to sort eight-bit unsigned binary numbers.

When the subroutine listed in Example 6-1 is called, register pair H must contain the starting address of the list, and register pair D must contain the node count, the number of eight-bit numbers that are contained in the list. The first instruction in ISORT saves the starting address of the list, which is contained in register pair H, on the stack. This 16-bit address is then decremented and complemented. The INXH instruction forms the 2s complement of the address just before the starting address of the list. If the starting address of the list is 040 100 (2080), what is contained in register pair H after the MOVLA instruction is executed? Register pair H contains 337 201 (DF81).

Example 6-1: A List-Sort Subroutine That Uses the Straight Insertion Method

```
/THIS SUBROUTINE USES THE SORT-BY-INSERTION
/ALGORITHM TO SORT SINGLE-PRECISION (8-BIT) UNSIGNED
/NUMBERS STORED IN MEMORY. ENTER THIS SUBROUTINE
/WITH THE STARTING ADDRESS OF THE LIST IN REGISTER
/PAIR H AND THE NODE COUNT IN REGISTER PAIR D.

ISORT,   PUSHH   /SAVE THE LIST ADDRESS ON THE STACK.
         DCXH    /DECREMENT THE ADDRESS BY ONE.
         MOVAH   /NOW FORM THE 2'S COMPLEMENT OF THIS
         CMA     /ADDRESS WHICH IS ONE LESS THAN
         MOVHA   /THE STARTING ADDRESS OF THE LIST.
         MOVAL   /THIS IS SO THE 8080 KNOWS THE
         CMA     /LOWER BOUNDARY OF THE LIST.
         MOVLA   /1'S COMPLEMENT IS IN REGISTER PAIR H.
         INXH    /NOW IT'S THE 2'S COMPLEMENT.
         SHLD    /SAVE THE 2'S COMPLEMENT OF THE
         BOTTM   /LOWER BOUNDARY OF THE LIST
         0       /IN BOTTM.
UP1,     POPH    /NOW GET THE ADDRESS OFF OF THE STACK.
         INXH    /INCREMENT THE ADDRESS.
         MOVBM   /GET A NODE (DATA VALUE) INTO B.
         DCXD    /DECREMENT THE NODE COUNT.
         MOVAD   /THEN SEE IF IT IS ZERO
         ORAE    /OR NOT. THIS INDICATES WHEN THE
         RZ      /ENTIRE LIST HAS BEEN SORTED.
         PUSHH   /SAVE THE INCREMENTED ADDRESS.
DOWN1,   DCXH    /DECREMENT THE LIST ADDRESS.
         PUSHD   /SAVE THE COUNT.
         XCHG    /GET THE LIST ADDRESS INTO D AND E.
         LHLD    /GET THE LOWER BOUNDARY ADDRESS.
         BOTTM
         0
         DADD    /ADD THE ADDRESS AND LOWEST ADDRESS.
         MOVAH   /IS THE RESULT ZERO?
         ORAL
         XCHG    /GET THE ADDRESS INTO H AND L.
         POPD    /POP THE COUNT OFF OF THE STACK.
         JZ      /YES, THE BEGINNING OF THE
         FARENF  /LIST HAS BEEN REACHED, SO SAVE
         0       /WHAT IS IN B IN THE LIST.
         MOVAM   /NOT AT THE BEGINNING, GET A NODE.
         CMPB    /IS IT LESS THAN B?
         JC      /YES, THEN SAVE B IN THE LIST.
         FARENF
         0
         INXH    /INCREMENT THE ADDRESS BY ONE.
         MOVMA   /SAVE THE NODE AT THE HIGHER ADDRESS.
         DCXH    /THEN DECREMENT THE ADDRESS.
         JMP     /AND SEE IF IT CAN BE MOVED DOWN
         DOWN1   /ANOTHER POSITION IN THE LIST.
         0
FARENF,  INXH    /THE NODE IN THE LIST IS SMALLER,
         MOVMB   /SO INCREMENT THE ADDRESS AND SAVE B.
         JMP     /SO LOOK AT THE NEXT CONSECUTIVE NODE
```

```
        UP1     /IN THE LIST AND SEE IF IT SHOULD BE
        0       /MOVED TO A LOWER ADDRESS.

BOTTM,  0       /THIS IS WHERE THE 2'S COMPLEMENT FOR
        0       /THE LOWEST ADDRESS IS STORED.
```

	Octal	Binary	Hex
Starting Address:	040 200	00100000 10000000	2080
Decremented to:	040 177	00100000 01111111	207F
Complemented to:	337 200	11011111 10000000	DF80
Incremented to:	337 201	11011111 10000001	DF81

This 16-bit result is then stored in R/W memory at BOTTM. This number will be used by the 8080 to determine when it has reached the beginning or bottom of the list.

To demonstrate how the ISORT subroutine operates, assume that the 8080 must sort the following list:

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	5	3	4	2	1

After the 8080 calculates the 2s complement of the last address below the beginning of the list, it pops the starting address of the list off of the stack into register pair H at UP1. Register pair H now contains the address of the memory location that contains the 5 in our example. This address is then incremented (to X+1) and the content of memory at this address (the 3) is moved from memory to the B register. The node count in register pair D (initially 005, because there are five nodes in the list) is then decremented and checked to see if a 0 result was obtained. Since the result is nonzero, the RZ instruction is not executed. Instead, the incremented address, which addresses the memory location (X+1) that contains the 3, is saved on the stack.

The content of register pair H is then decremented by 1 at DOWN1. Since the address in register pair H has been decremented, the 8080 has to determine if the content of register pair H is less than the starting address of the list. To do this, the node count in register pair D is saved on the stack, and the decremented address in register pair H is exchanged into register pair D. The 2s complement of the address that is 1 less than the starting address of the list is then loaded into register pair H, and the list address in register pair D is added to it. If the result of this addition is 0, the 8080 has decremented the list address past the beginning of the list. To check for this condition, register H is moved to the A register, and the content of the L register is ORed with the content of the A register. After the ORAL instruction is executed, the list address in register pair D is exchanged back into register pair

H, and the node count is popped off of the stack into register pair D. None of the instructions after the ORAL instruction affect any of the 8080 flags.

If the 0 flag is a logic 1, the 8080 jumps to FARENF. However, in our example, register pair H was decremented to memory address X, where the 5 is stored. Therefore, the 8080 does not jump to FARENF. Instead, it moves the 5 from memory location X to the A register, and this value is compared to the 3 (from X+1) in the B register. The 3 is less than the 5, so the JC to FARENF is not executed. Instead, the memory address in register pair H is incremented by 1 to X+1, and the 5 contained in the A register is stored in this memory location. At this point, two memory locations, X and X+1, both contain 5. Remember, the 3 is still stored in the B register. The address in register pair H is then decremented by 1 to X, and then the 8080 jumps back to DOWN1. The list now appears as follows:

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	5	5	4	2	1

(Register B contains the 3.)

At DOWN1, the address in register pair H is again decremented, to X−1. This is one less than the starting address of the list. When the 16-bit content of BOTTM is added to this address, the result is 0, so the 8080 jumps to FARENF. At FARENF, the address is incremented by 1 back up to X, and the content of the B register (3) is stored in this memory location. The nodes in the list are now arranged in the following order:

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	3	5	4	2	1

After the MOVMB instruction is executed, the 8080 jumps to UP1. As you can see, most of the instructions in this subroutine are used to ensure that the 8080 does not go beyond the end of the list (by means of the node count) or beyond the beginning of the list (by means of the 2s complement number stored in BOTTM).

At UP1, the list address is popped off of the stack and is incremented by 1 to X+2. Register pair H now addresses the memory location that contains the 4. The 4 is moved from memory to the B register, and the node count in the D register is decremented by 1 to 003. This is a nonzero result, so the 8080 saves address X+2 on the stack, decrements it to address X+1, and adds the content of BOTTM to it. The result of this addition is nonzero, so the 8080 loads the A register with the content of X+1 (5) and compares this to the 4 in the B register. The content of the B register is less than the content of the A register, so the 8080 has to write the 5 into the list where the 4 is stored. Therefore, the address in register pair H is incremented by 1 to X+2, and the 5 in the B register is stored in this memory location. Again, the 5 is contained in two memory locations used by the list.

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	3	5	5	2	1

(Register B contains the 4.)

After the 5 is written into memory location X+2, the address in register pair H is decremented to X+1, and the 8080 jumps to DOWN1. The memory address in register pair H is decremented by 1 to X. Since this address is not less than the starting address of the list, the 3 in memory location X is loaded into the A register and is compared to the 4 in the B register. The content of the B register is greater than the content of the A register, so the 8080 executes the JC to FARENF. At FARENF, the content of register pair H is incremented by 1 to X+1, and the 4 is saved in this memory location.

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	3	4	5	2	1

As you might guess, when the 2 is read from memory location X+3 into the B register, and sections of the ISORT subroutine are executed, the list appears as follows:

First we have

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	3	4	5	5	1

then

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	3	4	4	5	1

then

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	3	3	4	5	1

and finally,

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	2	3	4	5	1

When the 1 is read from memory location X+4 into the B register, the data values in the list are moved up one memory location, as follows:

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	2	3	4	5	5
Content:	2	3	4	4	5
Content:	2	3	3	4	5
Content:	2	2	3	4	5
Content:	1	2	3	4	5

In summary, this sorting method reads a node from the list and writes it into the B register of the 8080. This value is then compared to the next consecutive node at a *lower memory address.* If the node in the list is larger than the node in the B register, the node in the list is moved up one position (in this case, one memory location). The content of the B register is then compared to the node in the next lower memory location. Only when the node in the B register is larger than the node in the list is it written back into the list. However, as you saw in our example, the content of the B register is also written back into the list if the 8080 has "backed down" to the beginning or starting address of the list. In our example, the list contained no values less than 1, so this value was eventually stored at the beginning of the list.

Insertion Sort Best-Case/Worst-Case Sorting Times

Assume that the values 1, 5, 2, 7, 6, 8, 3, and 4 are stored in a list. Arrange these values so that the ISORT subroutine will require the least amount of time to sort the list. This time will be the *best-case time.* If the values are arranged from the lowest to the highest memory address as 1, 2, 3, 4, 5, 6, 7, and 8, the ISORT subroutine will require the least amount of time to sort the list. In other words, the list was already sorted when the ISORT subroutine was called. The *worst-case time* will occur when the values in the list are arranged in a descending order; for example, 8, 7, 6, 5, 4, 3, 2, and 1. This means that the ISORT subroutine will require the greatest amount of time to sort this list. Of course, the time required to sort most lists ("average" lists) will be between the best-case and worst-case times.

Exchange Sort (Bubble Sort)

Another method of sorting that we will discuss is an *exchange method,* as opposed to an insertion method, which was just discussed. The exchange method that we will discuss is called the *bubble-sort* method. Very simply, the bubble-sort method compares two nodes in a list. If the node stored in memory location X is greater than the node in memory location X+1, the nodes are exchanged. If the node in memory location X is less than the node in memory location X+1, no exchange takes place. This assumes that we want to sort the list in an ascending order, and that the nodes are single-precision (eight-bit values). This same assumption was used in the previous section of this chapter.

Regardless of whether or not an exchange takes place, the microcomputer then compares the nodes stored in memory locations X+1 and X+2. If the node in X+1 is greater than the node in X+2, the two nodes are exchanged. If the node in X+1 is less than the node in X+2, no exchange takes place. The 8080 then compares the nodes in memory locations X+2 and X+3. This compare-and-exchange process continues until the 8080 reaches the end of the list. *If any exchanges take place at any time, the 8080 sets a register or memory location to a specific value to indicate that an exchange did take place.*

After the entire list has been examined, the 8080 checks the content of this register or memory location to see if any exchanges took place. If an exchange did take place, this register or memory location is reset or initialized to a known value, and then the list is reexamined Only when the entire list has been examined, and *no* exchanges take place, will the list be sorted.

Table 6-2. Bubble Sorting a List of Data

Memory Address	Initially	1st Pass	2nd Pass	3rd Pass	4th Pass	5th Pass
X	3	3	2	2	2	1
X+1	2	2	3	3	1	2
X+2	5	5	4	1	3	3
X+3	4	4	1	4	4	4
X+4	1	1	5	5	5	5
		Result	**Result**	**Result**	**Result**	**Result**
X	3	2	2	2	1	1
X+1	2	3	3	1	2	2
X+2	5	4	1	3	3	3
X+3	4	1	4	4	4	4
X+4	1	5	5	5	5	5

Based on this description, where will the largest node (data value) be stored, at the beginning or at the end of the list? Within the boundaries of the list, the largest data value will be *bubbled-up* to the top of the list. This process of bubble-sorting data values in a list is shown in Table 6-2.

Initially, the list in Table 6-2 contains the following data values:

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	3	2	5	4	1

The first time the list is examined, the *first pass* through the list, the 3 and the 2 are compared. Since the 3 is greater than the 2, the two values are exchanged. Because an exchange has taken place, the exchange indicator is set. The 3 in memory location X+1 is then compared to the 5 in X+2. No exchange takes place because the 5 is greater than the 3. However, the 5 is then compared to the 4 in X+3. These two values are exchanged, and the exchange indicator is again set. The list now appears as follows:

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	2	3	4	5	1

The 5 in X+3 is compared to and exchanged with the 1 in X+4, and the exchange indicator is set a third time. As you can see, the 5 has been bubbled-up to the top of the list.

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	2	3	4	1	5

Since the entire list has now been examined, the 8080 checks the state of the exchange indicator. Since it is set, the 8080 clears it and then examines the list again, the *second pass* through the list. You can see the results of the second, third, and fourth passes through the list in Table 6-2. Finally, on the fifth pass through the list, none of the nodes have to be exchanged, so the exchange indicator remains cleared. The 8080 senses this, and returns to the program that called the bubble-sort subroutine. When the 8080 returns from the subroutine, the list is sorted.

A subroutine that sorts a list using the bubble-sort method is listed in Example 6-2. When BSORT is called, register pair H must be loaded with the starting address (the lowest memory address) of the list, and register pair D must be loaded with the final address (the highest memory address) of the list. The first six instructions in the BSORT subroutine subtract the starting address from the final address. The result is stored in register pair D. What do these instructions do? They have calculated the difference between the starting and final addresses, the node count.

In the ISORT subroutine (Example 6-1), the subroutine had to be called with the starting address in register pair H and the node count in register pair D. However, there may be cases where the starting and final addresses of the list are known, but the node count is not. Therefore, the first six instructions in the BSORT

Example 6-2: A List-Sort Subroutine That Uses the Exchange Method (Bubble Sort)

```
/THIS SUBROUTINE USES THE BUBBLE-SORT ALGORITHM
/TO SORT SINGLE-PRECISION (8-BIT) UNSIGNED
/NUMBERS STORED IN MEMORY. ENTER THIS SUBROUTINE
/WITH THE STARTING ADDRESS OF THE LIST
/(THE LOWER ADDRESS) IN REGISTER PAIR H AND THE
/FINAL ADDRESS IN REGISTER PAIR D.

BSORT,  MOVAE   /GET THE LO BYTE OF THE HIGHEST ADDRESS.
        SUBL    /SUBTRACT THE BEGINNING LO ADDRESS.
        MOVEA   /SAVE THE RESULT IN E.
        MOVAD   /GET THE HI BYTE OF THE HIGHEST ADDRESS.
        SBBH    /SUBTRACT THE BEGINNING HI ADDRESS.
        MOVDA   /SAVE THE RESULT IN D.
PASS1,  PUSHH   /SAVE THE INITIAL ADDRESS.
        PUSHD   /SAVE THE NODE COUNT ON THE STACK.
        MVIC    /THE C REGISTER IS USED TO INDICATE
        000     /IF AN EXCHANGE TAKES PLACE.
UP1,    MOVAM   /GET A NODE FROM THE LIST.
        INXH    /INCREMENT H AND L TO THE NEXT NODE.
        CMPM    /COMPARE A AND MEMORY.
        JC      /IF MEMORY > A, DO NOT EXCHANGE
        NEXT    /THEM. INSTEAD, EXAMINE THE NEXT TWO
        0       /NODES IN THE LIST.
        JZ      /IF THEY ARE EQUAL, DO NOT
        NEXT    /EXCHANGE THEM EITHER.
        0
        MOVBM   /SAVE THE CONTENT OF MEMORY IN B.
        MOVMA   /SAVE THE CONTENT OF A IN MEMORY.
        DCXH    /BACK UP ONE MEMORY LOCATION AND
        MOVMB   /THEN SAVE THE CONTENT OF B IN MEMORY.
        INXH    /INCREMENT THE LIST ADDRESS.
        MVIC    /THE C REGISTER IS USED TO INDICATE
        377     /THAT AN EXCHANGE HAS TAKEN PLACE.
NEXT,   DCXD    /DECREMENT THE COUNT.
        MOVAD   /IS THE COUNT 000?
        ORAE
        JNZ     /HAVE NOT CHECKED THE ENTIRE
        UP1     /LIST, SO KEEP CHECKING.
        0
        MOVAC   /GET THE EXCHANGE INDICATOR INTO
        ORAA    /THE A REGISTER AND SET THE FLAGS.
        POPD    /LOAD D AND E WITH THE NODE COUNT.
        POPH    /LOAD H AND L WITH THE INITIAL ADDRESS.
        JNZ     /AN EXCHANGE OCCURRED, SO EXAMINE
        PASS1   /THE LIST ONE MORE TIME.
        0
        RET     /RETURN WHEN NO EXCHANGES TAKE PLACE.
```

subroutine are used to calculate the node count, and this same sequence of instructions can also be used at the beginning of the ISORT subroutine. If these instructions are used in ISORT, an INXD instruction must be added to ISORT after the MOVDA instruction (Example 6-3).

Example 6-3. Calculating the Node Count in the ISORT Subroutine (Example 6-1)

```
/THIS SUBROUTINE USES THE SORT-BY-INSERTION
/ALGORITHM TO SORT SINGLE-PRECISION (8-BIT) UNSIGNED
/NUMBERS STORED IN MEMORY. ENTER THIS SUBROUTINE
/WITH THE STARTING ADDRESS OF THE LIST IN REGISTER
/PAIR H AND THE NODE COUNT IN REGISTER PAIR D.

ISORT,  MOVAE   /GET THE LSBY OF THE FINAL ADDRESS.
        SUBL    /SUBTRACT THE LSBY OF THE BEGINNING ADDRESS.
        MOVEA   /SAVE THE DIFFERENCE LSBY IN E.
        MOVAD   /GET THE MSBY OF THE FINAL ADDRESS.
        SBBH    /SUBTRACT THE MSBY OF THE BEGINNING ADDRESS.
        MOVDA   /SAVE THE DIFFERENCE MSBY IN D.
        INXD    /INCREMENT THE DIFFERENCE BY ONE.
        PUSHH   /SAVE THE LIST ADDRESS ON THE STACK.
        DCXH    /DECREMENT THE ADDRESS BY ONE.
        MOVAH   /NOW FORM THE 2'S COMPLEMENT OF THIS
        CMA     /ADDRESS WHICH IS ONE LESS THAN
        MOVHA   /THE STARTING ADDRESS OF THE LIST.
         •
         •
```

After the node count is calculated in the BSORT subroutine (Example 6-2), the starting address of the list in register pair H and the node count in register pair D are saved on the stack. The C register, which will be used as the "exchange indicator," is then cleared to 0. At UP1, the first node stored in the memory location (X) addressed by register pair H, is moved to the A register. The memory address in register pair H is then incremented by 1 to X+1, and the node in this memory location is compared to the content of the A register. If the content of memory is greater than the content of the A register, the 8080 jumps to NEXT. If this is the case, then the two nodes are not exchanged. If the two nodes are equal, the 8080 also jumps to NEXT.

If the content of memory at X+1 is less than the content of the A register, an exchange must take place. Therefore, the 8080 does not jump to NEXT. Assume that we are using the BSORT subroutine to sort the list shown in Table 6-2. The list is arranged as follows:

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	3	2	5	4	1

This means that the values 3 and 2 must be exchanged. Remember, the 3 is contained in the A register and the 2 is contained in the memory location addressed by register pair H (X+1). To perform the exchange, the 8080 moves the 2 in memory to the B register, and then saves the content of the A register (3) in the memory location addressed by register pair H (X+1). The list now contains the values

Memory Location:	X	X+1	X+2	X+3	X+4
Content:	3	3	5	4	1

After saving the 3 in memory location X+1, the memory address in register pair H is decremented and the 2 (in the B register) is written into memory location X. After the memory address is incremented back to X+1, the C register is loaded with the value 377 (FF) to indicate that an exchange has taken place during this pass through the list. After performing these exchange instructions, the 8080 executes the instructions at NEXT.

At NEXT, the 8080 determines if it has reached the top of the list. It does this by decrementing the node count in register pair D, and by checking for a "zero condition." Of course, since the DCXD instruction does not affect any of the 8080 flags, the MOVAD and ORAE instructions have to be executed to set or clear the flags based on the 16-bit number contained in register pair D. If the node count is nonzero, the 8080 jumps to UP1. In our example, the 8080 has only compared and exchanged the 3 and the 2 in the list. Therefore, the 8080 does jump to UP1.

When the 8080 executes the instructions at UP1 this time, it compares the 3 stored in memory location X+1 to the 5 stored in memory location X+2. No exchange takes place as a result of this comparison. However, during the first pass through the list, the 5 and the 4, and then the 1 and the 5, are exchanged. After the 8080 exchanges the 5 and the 1, the node count in register pair D is decremented to 0. Therefore, the 8080 does not jump to UP1 after the ORAE instruction is executed. Instead, it checks the state of the exchange indicator. If no exchanges have taken place, the C register contains 0. If at least one exchange has taken place, the C register contains 377 (FF). To determine the state of the exchange indicator, the content of the C register is moved to the A register, and the flags are set or cleared based on the content of the A register, when the ORAA instruction is executed.

The flags are set or cleared by the ORAA instruction, so the 8080 then pops the node count and the starting address of the list off of the stack. If the C register contains 377 (FF), which means that at least one exchange took place during the last pass through the list, the JNZ to PASS1 is executed. At PASS1, register pairs H and D are saved on the stack, and the C register is cleared to 0. The 8080 then examines the entire list again.

If the C register contains 0, then no exchanges took place during the last pass through the list. Therefore, the list is sorted and the 8080 returns from the BSORT subroutine. Based on this description, you should now know why the 8080 must make five passes through the list shown in Table 6-2. On the last and final pass, all of the nodes are in the proper order, so the exchange indicator is 0. The 8080 then returns from the BSORT subroutine.

Bubble Sort Best-Case/Worst-Case Sorting Times

As you would expect, the 8080 has to make between one and N passes through the list, where N is the number of nodes in the list. If the list is already sorted, the 8080 will only make one pass through the list when the BSORT subroutine is called. At the end of this pass, the C register will contain 0, since no exchanges took place. Therefore, the 8080 will return from the BSORT subroutine. If the nodes in the list are sorted in a completely reversed order, then the 8080 will have to make the maximum number (N) of passes through the list. This was the same case when we discussed the best-case and worst-case times for the insertion-sort method.

A Comparison of Insertion and Exchange Sorting

Both the ISORT and BSORT subroutines can be used to sort lists. The advantage of using one subroutine or the other is really determined by the *time* required to sort a list. To obtain some sorting times, we used the ISORT and BSORT subroutines to sort a list that contained 256 eight-bit unsigned numbers. The subroutines were used to sort (1) a list that was already sorted, (2) a list whose nodes were sorted in a reverse order, and (3) a list that contained all of the same numbers. The timing results are listed in Table 6-3.

Table 6-3. Sorting Times for the ISORT and BSORT Subroutines

List Nodes	ISORT	BSORT
Sorted	11.13 ms	3.41 ms
Reverse Order	1.05 s	1.26 s
All Equal	1.05 s	4.05 ms

From Table 6-3, we can see that the bubble sort is only faster than the insertion sort when the list is (1) already sorted, or (2) when the list contains all of the same numbers. Neither of these two cases is very common. When a large number of sorting operations must take place (either insertions or exchanges), as in the case of the list that contains numbers that are sorted in a reverse order, the insertion-sort (ISORT) subroutine requires less time (1.05 s) than the bubble-sort (BSORT) subroutine (1.26 s). Of course, most sorting times will be somewhere between the time required to sort a list that is already sorted and the time required to sort a list that is sorted in the reverse order. For a more detailed analysis of the time required by these two methods, refer to Knuth's work[1].

Limitations of the ISORT and BSORT Subroutines

What is the maximum size of the list that either of the sort subroutines (ISORT and BSORT) can be used to sort? The maximum size of the list is 65,536 eight-bit words (nodes). The reason for this is that register pair D is used to store the node count. Of course, both subroutines are limited to sorting eight-bit numbers. However, with a little ingenuity, both subroutines can be modified so that 16- or 24-bit numbers can be sorted. This means that 16- or 24-bit comparisons must be performed. Since the 8080 has no 16- or 24-bit compare instructions, two or three eight-bit subtractions will have to be performed. Of course, the subroutines can also be modified so that signed or floating-point numbers can also be sorted.

What will happen if you try to use these subroutines to sort data values that are stored in *R*ead-*O*nly-*M*emory (ROM)? We hope that you will never have the misfortune of trying this, but it may happen. If the ISORT subroutine is used, the 8080 will return from the subroutine without being able to change the order of the list, even though it will try. However, the 8080 will not return from the BSORT subroutine.

Remember, the ISORT subroutine makes only one pass through the list. When the 8080 gets to the end of the list in the BSORT subroutine, it checks the content of the C register to determine if any exchanges took place. If any did, the 8080 will make another pass through the list. Since the list is stored in ROM, it can never be sorted, so the 8080 will continue to make passes through the list. There is only one case where the 8080 will not remain in the BSORT subroutine, even if the list is stored in ROM. Do you know what this case is? If the data values in the list in ROM are already sorted, the 8080 will return from the BSORT subroutine after making only one pass through the list. The 8080 will do this because no exchanges took place.

SORTING ALPHANUMERIC STRINGS

You have now seen two different methods that can be used to sort numeric values—insertion sorting and exchange sorting. These same two methods can also be used to sort strings of alphanumeric characters. That is, alphabetize alphanumeric strings. However, we will only use the exchange sort (bubble sort) in our examples.

One difficulty in sorting strings composed of ASCII characters is that strings that contain 20 or 30 characters may have to be sorted, as opposed to our previous sorting examples where single-precision (eight-bit) data values were sorted. Now we are faced with the problem of sorting strings of characters that may not even be the same length. Suppose that we want the 8080 to sort the names listed in Table 6-4. As you can see from Table 6-4, three of the names contain five letters, two contain six letters, and one contains eight letters.

Table 6-4. A List That Contains Six Names

SMYTHE
SMITH
JONES
LEWIS
PETERSON
PETERS

To begin sorting (alphabetizing) the names listed in Table 6-4, we have to start by comparing the first two names. Since SMITH should come before SMYTHE in the list, the two names must be exchanged. Since the names contain a different number of characters (the strings are different "lengths"), it will be easiest to move the name SMYTHE to a temporary storage area. SMITH can then be moved to the memory locations used to store SMYTHE in the list, and then SMYTHE can be moved from the temporary storage area back to the list, immediately following SMITH.

Since we are using the bubble-sorted method, we will have to have an exchange indicator. The C register will be used to store this indicator. If the content of the C register is equal to 377 (FF) when the 8080 gets to the end of the list, at least one exchange took place during the last pass through the list. This means that the 8080 will have to make at least one more pass through the list. If the content of the C register is equal to 0, then the list is sorted and the 8080 can return from the subroutine.

For the subroutine to sort (alphabetize) the names, the strings in the list must have a *specific* format (*structure*). No strings can be added to the list unless this format is used. The format that we will use is defined as follows: Each string must have a 0 stored after the last alphanumeric character in the string. The 0 will be considered to be the last character in the string, the string terminator. If the string in the list is the last string, a 0 must be stored after the last character in the string, *and* a 233 (9B) must be stored in the list after this last string terminator. The 233 (9B) is the list terminator and it is the last character in the list. The subroutine that can be used to sort alphanumeric strings is listed in Example 6-4.

At the start of the ABSORT subroutine, the C register is loaded with 0 to indicate that no exchanges have taken place. Register pair H is then loaded with the starting address of the list, the address of the first character in the first string in the list. This same address is stored in memory at FIRST. The 8080 then starts to examine the list at UP1 for the first character in the second string. The 8080 will be able to find this character very easily, simply because it is stored in the list just after the first 0 (the termination character for the first string). Since 0 is not a printing ASCII character, it will not be contained in any names. Therefore, it is a good choice as the string terminator. Do not confuse the value 0 with the ASCII number 0 (260, B0). When the first character of the second string is found, the 8080 saves its memory address in SECOND. The 8080 now knows where the first characters of the first two strings are stored in memory. Therefore, just before AGAIN, the starting address of the second string is exchanged into register pair D, and register pair H is loaded with the starting address of the first string.

The 8080 now has to compare these two ASCII strings on a character-by-character basis. If the names in Table 6-4 are to be sorted, the 8080 will determine that SMYTHE should be after SMITH, so the position of the two names in the list must be exchanged. The 8080 will then compare SMYTHE to LEWIS, and then exchange them. The 8080 will continue this compare-and-exchange process until the entire list is sorted (alphabetized).

This means that at CMPNXT, the 8080 loads the A register with the first character contained in the second string. If a 0 is loaded into the A register, then the end of the second string has been found. If a 0 is read from memory, the 8080 checks to see if register pair H addresses the memory location that contains the 0 at the end of the first string. If the ends of *both* strings have been found, then the two strings are equal; for example, if the names PORTER and PORTER, or JONES and JONES, are being compared in the list. Therefore, no exchange should take place, so the 8080 jumps to SECBIG. By jumping to SECBIG, the 8080 can begin comparing the second and third strings in the list.

Suppose the 8080 finds the 0 at the end of the second string, but finds an alphanumeric character in the same "position" in the first string? Compare the strings ABS (the first string) to the string AB (the second string). The third character in the AB string is 0. However, the third character in the ABS string is S. Therefore, the two strings should be exchanged. Note that simply because one string is longer than another does not mean that they should be exchanged. If the two strings are equal on a character-by-character basis until the end of one string is found, and the second string is the shorter of the two, they should be exchanged. The 8080 exchanges the two strings by jumping to EXCH.

If the 8080 does not find the 0 at the end of the second string, it jumps to ENDLST, where the value read from the second

Example 6-4: Sorting Alphanumeric Strings Using the Exchange Method

```
ABSORT, MVIC    /SET THE EXCHANGE INDICATOR
        000     /TO ZERO (NO EXCHANGES).
        LXIH    /LOAD REGISTER PAIR H WITH THE
        LIST    /STARTING ADDRESS OF THE LIST
        0       /THAT CONTAINS ASCII STRINGS.
NEXT,   SHLD    /SAVE THIS ADDRESS AS THE
        FIRST   /STARTING ADDRESS OF THE FIRST
        0       /ASCII STRING.
UP1,    MOVAM   /GET A CHARACTER FROM THE STRING.
        INXH    /INCREMENT THE MEMORY ADDRESS.
        CPI     /FOUND THE END OF THE FIRST
        000     /STRING YET?
        JNZ     /NO, THEN KEEP LOOKING.
        UP1
        0
        SHLD    /YES, THEN SAVE THIS ADDRESS AS
        SECOND  /THE STARTING ADDRESS OF THE
        0       /SECOND ASCII STRING.
        XCHG    /PUT "SECOND" INTO D AND E.
AGAIN,  LHLD    /THEN LOAD H AND L WITH "FIRST."
        FIRST
        0
CMPNXT, LDAXD   /GET A CHARACTER FROM "SECOND."
        CPI     /END OF THE SECOND STRING?
        000
        JNZ     /NOT THE END OF THE SECOND
        ENDLST  /STRING, SO SEE IF IT IS THE END
        0       /OF THE LIST.
        CMPM    /000 AT THE END OF THE FIRST STRING ALSO?
        JZ      /YES, THEN THE STRINGS ARE EQUAL, DON'T
        SECBIG  /EXCHANGE THEM.
        0
        JMP     /OK, FOUND THE 000 AT THE END OF "SECOND"
        EXCH    /BUT NOT AT THE END OF "FIRST."
        0       /"FIRST" > "SECOND," SO EXCHANGE THEM.
ENDLST, CPI     /END OF THE LIST?
        233
        JZ      /YES, THEN SEE IF ANY EXCHANGES
        DONE    /TOOK PLACE THIS PASS THROUGH
        0       /THE LIST.
        CMPM    /COMPARE THE FIRST TO THE SECOND.
        JC      /"FIRST" IS GREATER THAN "SECOND"
        EXCH    /SO EXCHANGE THEM.
        0
        JNZ     /THEY ARE NOT EQUAL EITHER, SO
        SECBIG  /THE SECOND MUST BE GREATER THAN
        0       /THE FIRST, DON'T EXCHANGE THEM.
        INXH    /THE CHARACTER IN EACH STRING IS
        INXD    /EQUAL, SO TEST THE NEXT TWO CON-
        JMP     /SECUTIVE  CHARACTERS.
        CMPNXT
        0
DONE,   MOVAC   /GET THE EXCHANGE INDICATOR.
        ORAA    /HAVE ANY EXCHANGES TAKEN PLACE?
```

```
         RZ       /NO EXCHANGES, THE LIST IS SORTED.
         JMP      /THE LIST IS NOT SORTED, TRY
         ABSORT   /AGAIN.
         0
EXCH,    MVIC     /SET THE EXCHANGE INDICATOR
         377      /TO 377.
         LHLD     /LOAD REGISTER PAIR H WITH THE ADDRESS
         FIRST    /FOR THE FIRST ASCII CHARACTER STRING.
         0
         XCHG     /PUT THE ADDRESS INTO D AND E.
         LXIH     /LOAD REGISTER PAIR H WITH A R/W
         TMP      /MEMORY ADDRESS WHERE A STRING
         0        /CAN BE TEMPORARILY STORED.
         CALL     /MOVE FROM D AND E TO H AND L.
         MOVE     /RETURN WITH D AND E POINTING TO
         0        /THE SECOND STRING.
         LHLD     /LOAD REGISTER PAIR H WITH
         FIRST    /THE ADDRESS FOR "FIRST."
         0
         CALL     /NOW MOVE D AND E TO H AND L, MOVE "SECOND"
         MOVE     /TO THE SECTION OF R/W MEMORY
         0        /THAT WAS USED TO STORE "FIRST."
         SHLD     /SAVE H AND L AS THE NEW SECOND STRING
         SECOND   /MEMORY ADDRESS.
         0
         LXID     /H AND L POINT TO "SECOND," SO LOAD
         TMP      /D AND E WITH THE TEMPORARY LO-
         0        /CATIONS WHERE "FIRST" IS.
         CALL     /NOW MOVE "FIRST," STORED AT TMP,
         MOVE     /TO THE OLD SECOND ADDRESS.
         0
SECBIG,  LHLD     /THEN LOAD H AND L WITH THE NEW
         SECOND   /SECOND STRING MEMORY ADDRESS.
         0
         JMP      /THEN COMPARE THESE TWO STRINGS.
         NEXT
         0
MOVE,    LDAXD    /GET A VALUE FROM MEMORY (D AND E).
         MOVMA    /SAVE THE VALUE (H AND L).
         INXH     /INCREMENT THE ADDRESS IN H AND L.
         INXD     /THEN INCREMENT THE ADDRESS IN D AND E.
         CPI      /WAS THE STRING TERMINATOR JUST
         000      /MOVED?
         JNZ      /NO, THEN MOVE ANOTHER CHARACTER.
         MOVE
         0
         RET      /THE TERMINATOR WAS MOVED, SO RETURN.
```

string is compared to the list terminator, 233 (9B). If this value was read from memory, the 8080 has reached the end of the list. Therefore, it must examine the content of the C register to determine if any exchanges took place during the last pass through the list. The 8080 examines the content of the C register when it jumps to DONE. If the C register contains 0, the 8080 jumps back to ABSORT so that it can examine the strings in the list again.

If the 8080 finds neither a 0 nor a 233 (9B), it compares the character in the first string addressed by register pair H to the character read into the A register from the memory location addressed by register pair D (a character from the second string). If the character in the first string has an ASCII value that is greater than the character in the same "position" in the second string, they must be exchanged. The JC to EXCH, just after the CMPM instruction, will cause the two strings to be exchanged. If the character in the first string is less than the character in the second string, then the two strings are in the proper order. This situation will occur if the string, SMITH (the first string), is compared to the second string, SMYTHE. The value for ASCII I is less than the value for ASCII Y. By jumping to SECBIG, the 8080 can begin the comparison process between the second and third strings in the list.

If the two characters in the strings are equal, for example, when the S in SMYTHE (first string) is compared to the S in SMITH (second string), the 8080 simply increments the addresses contained in register pairs H and D and jumps back to CMPNXT. This same sequence of instructions will be performed twice when SMITH is compared to SMYTHE, once when the letters S are compared and once when the letters M are compared. Note that the 8080 can only get "out of" the CMPNXT loop when one of five situations occurs; (1) when the 0 at the end of the second string is found, but the 0 at the end of the first string is not found (an exchange will take place), (2) when the 0s at the end of both the first and second strings are found (no exchange will take place), (3) when the 233 (9B) at the end of the list is found (no exchange will take place), (4) when a character in the first string is found to be greater than the character in the same position in the second string (an exchange will take place), or (5) when a character in the first string is found to be less than the character in the same position in the second string (no exchange will take place).

The EXCH section of the subroutine actually performs the exchange operation. There are four basic tasks that this section of the subroutine performs. First, the C register is loaded with 377 (FF) to indicate that an exchange has taken place (although when the C register is set, the actual exchange really has not been performed yet). Of course, when the 8080 reaches the end of the list, it will examine the content of the C register to determine if the list is sorted. After loading the C register with 377 (FF), the 8080 moves the first string, the string addressed by register pair H, to a series of R/W memory locations, starting at TMP. The second string is then loaded into the section of memory within the list that was used to store the first string. Finally, the 8080 moves the first string from TMP back into the list immediately after the second string, which was just moved. The positions of the first and second strings are now reversed. The second string is in the position originally occupied by the first string, and the first string is in the position that was originally occupied by the second string.

Once the strings have been exchanged, register pair H is loaded with the memory address where the second of the two strings is stored in memory, and then the 8080 jumps to NEXT. This is done so that the 8080 can compare the second and third strings in the list. If the names listed in Table 6-4 are to be sorted, then the EXCH section of the ABSORT subroutine will be executed when the Y in SMYTHE is compared to the I in SMITH. After exchanging the two strings, the 8080 will next compare SMYTHE to JONES. These two strings will also be exchanged. In fact, SMYTHE will be "bubbled-up" to the top of the list.

Do not become confused by the use of the symbolic addresses FIRST and SECOND in the ABSORT subroutine (Example 6-4). When the subroutine is first called, FIRST is used to store the starting address of the first string, and SECOND is used to store the starting address of the second string. However, if a list that contains 50 strings is being sorted, then at some point FIRST will be used to store the starting addresses of the 23rd, 32nd, and 41st strings. At the same time, SECOND will be used to store the starting addresses of the 24th, 33rd, and 42nd strings.

Demonstrating the String-Sort (ABSORT) Subroutine

A simple demonstration program can be written so that the ABSORT subroutine can be tested. This program is listed in Example 6-5. By executing this program, ASCII character strings can be entered into the microcomputer and stored in R/W memory. Once all of the strings have been entered, they are sorted and then the sorted list is printed on a teletypewriter or crt.

In Example 6-5, the stack pointer is loaded with a R/W memory address, and then a carriage return and line feed are printed on the teletypewriter or crt. Register pair H is loaded with the R/W memory address where the strings will be stored when the 8080 executes the LXIH instruction. At CHARIN, the TTYIN subroutine is called so that an ASCII character can be entered into the microcomputer from the teletypewriter or crt. When a key is pressed, the character is also printed. The 8080 returns from the TTYIN subroutine with the seven-bit ASCII character in the A register. If the RETURN key is pressed, or a CTRL/C is generated, the 8080 performs some special operations. Otherwise, the ASCII value is stored in the R/W memory location addressed by

Example 6-5: A Demonstration Program for the Alphanumeric String-Sort Subroutine (ABSORT)

```
START,   LXISP    /LOAD THE STACK POINTER WITH A
         STACK    /R/W MEMORY ADDRESS.
         0
         CALL     /PRINT A CARRIAGE RETURN AND A LINE
         CRLF     /FEED ON THE TELETYPEWRITER OR CRT.
         0
         LXIH     /LOAD REGISTER PAIR H WITH THE STARTING
         LIST     /ADDRESS (R/W MEMORY) OF THE LIST.
         0
CHARIN,  CALL     /GET A CHARACTER FROM THE TTY OR CRT.
         TTYIN
         0
         CPI      /WAS A CTRL/C ENTERED?
         003
         JZ       /YES, THEN SAVE A 233 (9B) AT THE END
         SORT     /OF THE LIST AND SORT IT.
         0
         CPI      /A CTRL/C WAS NOT ENTERED, WAS A
         015      /CARRIAGE RETURN?
         JZ       /YES, THEN SAVE A STRING TERMINATOR
         ENDLN    /OF ZERO AT THE END OF THE STRING.
         0
         MOVMA    /NEITHER A RETURN NOR A CTRL/C, SAVE THE VALUE
         INXH     /IN MEMORY AND THEN INCREMENT THE LIST ADDRESS.
         JMP      /THEN INPUT ANOTHER ASCII CHARACTER.
         CHARIN
         0
ENDLN,   MVIM     /A CARRIAGE RETURN WAS ENTERED, SAVE A ZERO
         000      /AFTER THE STRING IN MEMORY.
         INXH     /INCREMENT THE LIST ADDRESS.
         CALL     /PRINT A CARRIAGE RETURN AND A LINE
         CRLF     /FEED AFTER THE STRING.
         0
         JMP      /THEN GET ANOTHER ASCII CHARACTER.
         CHARIN
         0
SORT,    MVIM     /A CTRL/C WAS ENTERED, SO TERMINATE THE
         233      /LIST WITH A 233 (9B).
         CALL     /NOW SORT THE ASCII STRINGS STORED
         ABSORT   /IN THE LIST.
         0
         LXIH     /THE LIST IS SORTED, LOAD REGISTER PAIR
         LIST     /H WITH THE STARTING ADDRESS OF THE LIST.
         0
LINE,    CALL     /FIRST, PRINT A CARRIAGE RETURN
         CRLF     /AND A LINE FEED.
         0
PRINT,   MOVAM    /GET A CHARACTER FROM THE LIST.
         CPI      /IS IT A STRING TERMINATOR?
         000
         JZ       /YES, THEN PRINT A CARRIAGE RETURN
         ENDS     /AND A LINE FEED AFTER THE STRING IS
         0        /PRINTED ON THE TELETYPEWRITER OR CRT.
```

```
        CPI     /NOT A STRING TERMINATOR, IS IT A LIST
        233     /TERMINATOR?
        JNZ     /NO, THEN PRINT THE CHARACTER OUT.
        NOTEND
        0
        HLT     /FOUND THE LIST TERMINATOR, SO HALT.
NOTEND, CALL    /NOT THE END OF THE STRING OR THE LIST,
        TTYOUT  /SO PRINT THE CHARACTER ON THE TTY OR CRT.
        0
        INXH    /INCREMENT THE LIST ADDRESS.
        JMP     /THEN GET ANOTHER CHARACTER FROM THE LIST
        PRINT   /AND CHECK ITS VALUE.
        0
ENDS,   CALL    /THE ZERO AT THE END OF THE STRING WAS
        CRLF    /FOUND, SO PRINT A CR AND AN LF.
        0
        INXH    /THEN INCREMENT H AND L PAST THE ZERO,
        JMP     /AND GET ANOTHER CHARACTER FROM THE LIST.
        PRINT
        0
CRLF,   MVIA    /LOAD THE A REGISTER WITH THE ASCII
        215     /VALUE FOR THE CARRIAGE-RETURN CHARACTER.
        CALL    /AND THEN PRINT THE CHARACTER ON THE
        TTYOUT  /TELETYPEWRITER OR CRT.
        0
        MVIA    /THEN LOAD THE A REGISTER WITH THE ASCII
        212     /VALUE FOR THE LINE-FEED CHARACTER.
        JMP     /AND PRINT IT ON THE TTY OR CRT.
        TTYOUT
        0
TTYIN,  IN      /INPUT THE UART'S FLAG WORD.
        001
        ANI     /SAVE ONLY THE RECEIVER'S FLAG IN THE
        001     /A REGISTER.
        JZ      /THE FLAG IS ZERO, SO WAIT FOR IT
        TTYIN   /TO BECOME A LOGIC ONE.
        0
        IN      /THE FLAG IS A LOGIC ONE, INPUT THE
        000     /ASCII CHARACTER.
        ANI     /SET THE PARITY BIT TO A LOGIC ZERO
        177     /(BIT D7).
TTYOUT, PUSHPSW /SAVE THE ASCII CHARACTER ON THE STACK.
TTY1,   IN      /INPUT THE UART'S FLAG WORD.
        001
        ANI     /SAVE ONLY THE TRANSMITTER'S FLAG IN
        004     /THE A REGISTER.
        JZ      /THE FLAG IS A LOGIC ZERO, SO WAIT
        TTY1    /FOR IT TO BECOME A LOGIC ONE.
        0
        POPPSW  /THE FLAG IS A LOGIC ONE, GET THE
        OUT     /ASCII CHARACTER INTO REGISTER A AND
        000     /OUTPUT IT TO THE UART.
        RET
```

register pair H. The memory address is then incremented and the JMP to CHARIN is executed so that another ASCII value can be entered.

The ASCII values for the RETURN and CTRL/C are both used as termination values. If the RETURN value is entered, the 8080 treats all of the ASCII values entered since the last RETURN value as a single string. Therefore, when a RETURN value is entered, the 8080 jumps to ENDLN, where a 0 is stored in memory just after the last character in the string. This means that a string is composed of a *single line of ASCII characters.*

The CTRL/C is used to indicate to the microcomputer when all of the strings have been entered into the list. This means that when the last string of characters for the list has been typed in, it must be terminated with a RETURN, and then the entire list must be terminated with a CTRL/C. The CTRL/C will cause a 233 (9B) to be saved after the 0 that is stored in the list after the last string of ASCII characters. The reason that the 0 and 233 (9B) must be stored in memory is simply because this is the list format required by the ABSORT subroutine. After the CTRL/C is entered and the 233 (9B) is stored at the end of the list, the 8080 calls the ABSORT subroutine so that the list is sorted (alphabetized).

Example 6-6: Using the Demonstration Program With Some Sample Strings

(A) Unsorted Strings That Were Entered on the Teletypewriter

```
DAVE
NANCY
CHRIS
JON
JANE
SARA
LISA
DAVID
BETH
TYCHON, INC.
HOWARD W. SAMS
```

(B) The Sorted Strings That Were Printed on the Teletypewriter

```
BETH
CHRIS
DAVE
DAVID
HOWARD W. SAMS
JANE
JON
LISA
NANCY
SARA
TYCHON, INC.
```

When the 8080 returns from the ABSORT subroutine, the list is sorted, so the 8080 prints the sorted content of the list on the teletypewriter or crt. As the strings are read from memory (LINE), they are printed on the teletypewriter or crt. The 8080 checks the list values as they are read from memory for both a 233 (9B) and 0. If a 0 is read from memory, the entire ASCII string has been printed. Therefore, the 8080 calls the CRLF subroutine, so that a carriage return and a line feed are printed on the teletypewriter or crt after the string. If a 233 (9B) is read from memory, the 8080 halts because it has reached the end of the list and all of the strings have been printed. Example 6-6 contains some strings that we entered into the microcomputer and the sorted result that was printed out.

What will happen if the ABSORT subroutine is used to sort strings that contain both letters and numbers? Suppose a list is created that contains the two strings AAA and AA1. Once the list is sorted, which string will be first in the list and which will be second? The string AA1 will be the first string in the list, and AAA will be second. This means that ABSORT can be used to sort *alphanumeric strings* (ASCII letters and numbers). Since the values 0 and 233 (9B) are used as termination characters, the ABSORT subroutine can also be used to sort strings that contain punctuation!

REFERENCES

1. Knuth, D. E. *The Art of Computer Programming. Volume 3, Sorting and Searching.* Addison-Wesley Publishing Co., Reading, MA, 1973.
2. Wirth, N. *Algorithms + Data Structures = Programs.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1976.
3. Horowitz, E. and Sahni, S. *Fundamentals of Data Structures.* Computer Science Press, Inc., Woodland Hills, CA, 1976.

7

Look-Up Tables

In Chapter 5 of *8080/8085 Software Design—Book 1*[1], we described a number of mathematical subroutines. We mentioned that these subroutines could be used to convert data values from one *data domain* to another *data domain.* This means that a temperature expressed in degrees Fahrenheit can be converted to a temperature expressed in degrees Celsius, or a length in feet can be converted to a length in centimeters by using one of these subroutines. The relationship between degrees in Fahrenheit and Celsuis is well known,

$$°C = \frac{5}{9} \times (°F - 32) \text{ or } °F = \frac{9}{5} \times °C + 32$$

Multiplication, division, subtraction, and addition subroutines can be used for this conversion. A look-up table can also be used to convert a number in one data domain (for example, degrees Fahrenheit) to another data domain (for example, degrees Celsius).

A *look-up table* is simply *a collection of nodes stored in sequential memory locations.* We have used the term *nodes*, because a look-up table may contain data and/or addresses. To convert degrees Fahrenheit to degrees Celsius using a look-up table, the temperature in degrees Fahrenheit would be used as part of the address that addresses a memory location in the look-up table that contains the appropriate temperature in degrees Celsius. To address or "point" to the correct temperature stored in the table, the temperature in degrees Fahrenheit would be added to the *base address* or starting address of the look-up table. The base address is simply the address of the first memory location used to store the first node (or part of the first node) in the look-up table. The result of adding the temperature to the base address will "generate" a suitable address. Of course, this whole procedure assumes that the base address is the lowest memory address, and that the remainder of the look-up table is stored in sequential memory locations at higher addresses.

Look-up tables can be used in a number of software solutions. Assemblers, BASIC interpreters, and BASIC compilers use look-up tables. An assembler might use a look-up table to determine the op code for the instruction represented by the ASCII characters J, M, and P that are stored in consecutive memory locations. For an assembler, this look-up table is often called a *symbol table* because it contains values (op codes) for a number of symbols (really ASCII strings; the mnemonics). A BASIC interpreter could use a look-up table so that it executes the proper instruction sequence when a LET, FOR, IF, DIM, or READ statement is encountered in a user's program (written in the BASIC language). Some additional uses for look-up tables are summarized in Table 7-1. In the remainder of this chapter, we will describe the organization of look-up tables, the application of look-up tables, and their "software drivers."

One of the look-up table applications listed in Table 7-1 is not very practical. Do you know which one it is? Look-up tables are not generally used to convert pounds to kilograms, miles to kilometers, or binary to BCD. The reason for this is simply that there are *simple* equations that can be used to convert from one unit of measure (data domain) to another unit of measure (data domain). For instance, 1 pound = 0.454 kilogram and 1 mile = 1.6 kilometers. On the other hand, there are no simple equations that can be used to convert ASCII characters to the appropriate dots and dashes for Morse code. From Chapter 5 of *8080/8085 Software Design—Book 1*[1], we know that the sine of an angle can be approximated by using the formula

$$\text{sine}(X) = X - \frac{X^3}{3!} + \frac{X^5}{5!} - \frac{X^7}{7!} + \frac{X^9}{9!}$$

However, to perform these calculations may require a millisecond or two. By using a look-up table, we can obtain the sine of any

Table 7-1. Possible Applications for Look-Up Tables

1. Converting pounds to kilograms, miles to kilometers, or binary to BCD.
2. Assembler, interpreter, or compiler string conversion.
3. Convert BCD numbers to seven-segment bar codes (for LED displays).
4. Convert ASCII to Morse code and vice versa.
5. Ciphering and deciphering messages.
6. Finding different valid codes for electronic locks.
7. Relating an SSAN to a person's name.
8. Calculating the sine of an angle or the logarithm of a number.

angle between 0° and 360° (the angle may only be an integer value) within 114 μs.

Why are look-up tables used? Look-up tables are generally used in only two situations; (1) when there is no simple relationship between the two data domains, and (2) when the conversion must be performed as quickly as possible. Possible conversion problems include converting an ASCII character to a Morse code character, an angle to the sine of the angle, or a BCD number to a seven-segment LED bar code.

To determine the sine of an angle, a look-up table can be used. In fact, you can even go to integrated-circuit distributors and *purchase* a sine look-up table! This table is contained in *R*ead-*O*nly-*M*emory (ROM), and the ROM can be wired to your microcomputer system just like any other memory integrated circuit. The look-up table content of the National Semiconductor Corporation MM5220BM ROM is shown in Fig. 7-1. The "address" of each sine in the table is really the angle expressed in either radians or degrees. The contents of each memory location is the sine of this angle. Note that the addresses listed in Fig. 7-1 are in decimal, but they can be easily converted to octal or hexadecimal. Observe that the B_1 output of the ROM is the MSB of the sine, and that the B_8 output is the LSB of the sine. Also remember that the sine of any angle between 0° and 90° is between 0 and 1. Therefore, the significance of the output bits is as follows: $B_1 = 1/2$, $B_2 = 1/4$, $B_3 = 1/8$, $B_4 = 1/16$, $B_5 = 1/32$, $B_6 = 1/64$, $B_7 = 1/128$, and $B_8 = 1/256$.

One of the difficulties in using this device is that there are 128 values for the sine of the angles between 0° and 90°. Therefore, each memory location represents a change in the angle of 0.703°. This is not a particularly easy multiple to work with. To determine the sine of an angle using this look-up table, the proper address for the ROM would be calculated as follows:

$$\left(\frac{X}{90_{10}}\right) \times 128_{10} = \text{memory address}$$

This means that for $X = 45°$ the memory address is 64_{10}, and for $X = 30°$ the memory address is 43_{10}. By calculating the proper address, the sine of the angle can be determined. This calculation may require some multiplication and division, but it will still take very little time to find the sine of 30° or the sine of 71°.

To simplify the process of determining the sine of an angle, it would be easier to use a look-up table where every address represents an increment of 1° rather than 0.703°. To generate this new look-up table, a program was written in the BASIC language. The sine look-up table that this program generated is listed in Table 7-2. The most-significant bit of the binary values in this sine look-up table is 0. This was done so that a sign bit can be included in the sine of the angle after the 8080 reads the sine value from the table. Therefore, the format (sign and magnitude, not 2s complement) of the eight-bit words in the table is as follows:

S.XXXXXXX

The most-significant bit (S) of the eight-bit word is used for the sign of the seven-bit sine value. Bits XXXXXXX represent the seven-bit sine value of the angle. Since the binary point is between bits D_7 and D_6, bit D_6 is assigned the significance of 1/2, $D_5 = 1/4$, $D_4 = 1/8$, $D_3 = 1/16$, $D_2 = 1/32$, $D_1 = 1/64$, and $D_0 = 1/128$. If you compare these seven-bit sine values to the eight-bit sine values in the MM5220BM sine look-up table ROM (Fig. 7-1), you will see good agreement. Remember, the angles differ by 0.703° in the ROM and by 1° in the computer-generated look-up table (Table 7-2).

Using the preceding format, what would the sine of 30° look like? The sign of the sine is positive, and the sine of 30° is 0.5. Therefore, the sine of 30° is 01000000. The sine of 90° is 1, and the sign is positive, so the value that the computer must generate would be 01111111. The sine of 181° is negative, but the absolute value for the sine of 181° is equal to the sine of 1°. The sine of 1° is 0.0175, or 00000010 (1/64 or 0.015625). Therefore, the sine of 181° is 10000010. Remember, when dealing with fractional binary numbers, we may not get a particularly accurate result. Using the look-up table in Table 7-2, we must not expect the sine of the angle to be more accurate than 1/128, or 0.0078125 (0.78125% accuracy). This means that the sine of the angle may be accurate at most by ±0.78125%. This is in agreement with the value obtained for the sine of 1° $(0.015625 + 0.0078125 = 0.0234375)$. Therefore, the sine of 1° is accurate to ±0.78125%.

Of course, some software instructions must be executed so that the data values in the sine look-up table can be accessed. Perhaps the simplest method to use would be to call a subroutine, with the angle in binary degrees in the A register. This is what must be done if the SINANG subroutine in Example 7-1 is called.

In Example 7-1, the 8080 first compares the angle in the A register to the immediate data byte 133 (decimal 91). If the angle in the A register is equal to or greater than 91, the 8080 returns from the subroutine with the carry cleared to a logic 0. Immediately following the call to the SINANG subroutine, the user may want to store a JNC to ERROR, where ERROR is a subroutine that notifies the user that the microcomputer attempted to determine the sine of an angle that was greater than 90°. If the content of the A register

Example 7-1: Calculating the Sine of an Angle Between 0° and 90° Using a Sine Look-Up Table

```
/THIS SUBROUTINE CALCULATES THE SINE OF THE BINARY ANGLE
/CONTAINED IN THE A REGISTER. THE ANGLE MUST BE BETWEEN
/0 AND 90 DEGREES. IF IT IS NOT, THE CARRY WILL BE A
/LOGIC ZERO WHEN THE 8080 RETURNS FROM THE SUBROUTINE.
/IF THE ANGLE IS "VALID," THE SINE OF THE ANGLE WILL BE CON-
/TAINED IN THE A REGISTER WHEN THE 8080 RETURNS.

SINANG, CPI     /COMPARE THE ANGLE TO 91 DEGREES
        133     /(OCTAL 133 = DECIMAL 91).
        RNC     /THE ANGLE IS TOO LARGE, RETURN.
        PUSHH   /SAVE REGISTER PAIR H
        PUSHB   /AND REGISTER PAIR B ON THE STACK.
        LXIH    /LOAD REGISTER PAIR H WITH THE
        SINTAB  /BASE ADDRESS (STARTING ADDRESS) OF THE
        0       /SINE LOOK-UP TABLE.
        MVIB    /SET THE MSBY OF REGISTER PAIR B TO
        000     /ZERO.
        MOVCA   /MOVE THE ANGLE TO THE LSBY OF REGISTER
        DADB    /PAIR B. ADD THE ANGLE TO THE ADDRESS.
        MOVAM   /GET THE SINE OF THE ANGLE FROM THE TABLE.
        POPB    /POP REGISTER PAIR B
        POPH    /AND REGISTER PAIR H OFF OF THE STACK.
        STC     /SET THE CARRY.
        RET

SINTAB, 000     /THIS IS THE SINE OF ZERO DEGREES.
         •      /THE REMAINDER OF THE SINE LOOK-UP
         •      /TABLE IS STORED HERE.
         •
```

is between 0° and 90°, the 8080 will not execute the RNC instruction. Instead, register pairs H and B are saved on the stack.

The LXIH instruction loads register pair H with the base address of the look-up table, and the MVIB instruction loads 0 into the MSBY of register pair B. The angle in the A register is then moved to the LSBY of register pair B (the C register). The 8080 adds the angle in register pair B to the base address of the look-up table in register pair H. The resulting *address* is left in register pair H. The sine of the angle is then read from the look-up table into the A register. Register pairs B and H are then popped off of the stack and the carry is set to a logic 1 by the STC instruction. This instruction must be executed so that the JNC to ERROR is not executed when the 8080 returns from the SINANG subroutine. Of course, the JNC to ERROR does not have to be stored in memory after the call to SINANG. The 8080 then returns from SINANG with the sine of the angle in the A register.

As you can see, the 8080 creates a memory address by adding the angle to the base address of the look-up table. The sine of the angle can then be read from the memory location addressed by

ADDRESS REFERENCE	FUNCTION		CODE							
	INPUT		OUTPUT							
	DEGREES	RADIANS	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1
0	.00	.000	0	0	0	0	0	0	0	0
1	.70	.012	1	1	0	0	0	0	0	0
2	1.41	.025	0	1	1	0	0	0	0	0
3	2.11	.037	1	0	0	1	0	0	0	0
4	2.81	.049	0	0	1	1	0	0	0	0
5	3.52	.061	1	1	1	1	0	0	0	0
6	4.22	.074	1	1	0	0	1	0	0	0
7	4.92	.086	0	1	1	0	1	0	0	0
8	5.63	.098	1	0	0	1	1	0	0	0
9	6.33	.110	0	0	1	1	1	0	0	0
10	7.03	.123	1	1	1	1	1	0	0	0
11	7.73	.135	0	1	0	0	0	1	0	0
12	8.44	.147	1	0	1	0	0	1	0	0
13	9.14	.160	0	0	0	1	0	1	0	0
14	9.84	.172	0	0	1	1	0	1	0	0
15	10.55	.184	1	1	1	1	0	1	0	0
16	11.25	.196	0	1	0	0	1	1	0	0
17	11.95	.209	1	0	1	0	1	1	0	0
18	12.66	.221	0	0	0	1	1	1	0	0
19	13.36	.233	1	1	0	1	1	1	0	0
20	14.06	.245	0	1	1	1	1	1	0	0
21	14.77	.258	1	0	0	0	0	0	1	0
22	15.47	.270	0	0	1	0	0	0	1	0
23	16.17	.282	1	1	1	0	0	0	1	0
24	16.88	.295	0	1	0	1	0	0	1	0
25	17.58	.307	1	0	1	1	0	0	1	0
26	18.28	.319	0	0	0	0	1	0	1	0
27	18.98	.331	1	1	0	0	1	0	1	0
28	19.69	.344	0	1	1	0	1	0	1	0
29	20.39	.356	1	0	0	1	1	0	1	0
30	21.09	.368	0	0	1	1	1	0	1	0
31	21.80	.380	1	1	1	1	1	0	1	0
32	22.50	.393	0	1	0	0	0	1	1	0
33	23.20	.405	1	1	1	0	0	1	1	0
34	23.91	.417	1	1	1	0	0	1	1	0
35	24.61	.430	0	1	0	1	0	1	1	0
36	25.31	.442	1	0	1	1	0	1	1	0
37	26.02	.454	0	0	0	0	1	1	1	0
38	26.72	.466	1	1	0	0	1	1	1	0
39	27.42	.479	0	1	1	0	1	1	1	0
40	28.13	.491	0	0	0	1	1	1	1	0
41	28.83	.503	1	1	0	1	1	1	1	0
42	29.53	.515	0	1	1	1	1	1	1	0
43	30.23	.528	0	0	0	0	0	0	0	1
44	30.94	.540	1	1	0	0	0	0	0	1
45	31.64	.552	0	1	1	0	0	0	0	1
46	32.34	.565	1	0	0	1	0	0	0	1
47	33.05	.577	1	1	0	1	0	0	0	1
48	33.75	.589	0	1	1	1	0	0	0	1
49	34.45	.601	1	0	0	0	1	0	0	1
50	35.16	.614	1	1	0	0	1	0	0	1
51	35.86	.626	0	1	1	0	1	0	0	1
52	36.56	.638	0	0	0	1	1	0	0	1
53	37.27	.650	1	1	0	1	1	0	0	1
54	37.97	.663	1	0	1	1	1	0	0	1
55	38.67	.675	0	0	0	0	0	1	0	1
56	39.37	.687	0	1	0	0	0	1	0	1
57	40.08	.699	1	0	1	0	0	1	0	1
58	40.78	.712	1	1	1	0	0	1	0	1
59	41.48	.724	1	0	0	1	0	1	0	1
60	42.19	.736	0	0	1	1	0	1	0	1
61	42.89	.749	0	1	1	1	0	1	0	1
62	43.59	.761	0	0	0	0	1	1	0	1
63	44.30	.773	1	1	0	0	1	1	0	1

Fig. 7-1. The look-up table contained in a National

ADDRESS REFERENCE	FUNCTION		CODE							
	INPUT		OUTPUT							
	DEGREES	RADIANS	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1
64	45.00	.785	1	0	1	0	1	1	0	1
65	45.70	.798	1	1	1	0	1	1	0	1
66	46.41	.810	1	0	0	1	1	1	0	1
67	47.11	.822	1	1	0	1	1	1	0	1
68	47.81	.834	1	0	1	1	1	1	0	1
69	48.52	.847	0	0	0	0	0	0	1	1
70	49.22	.859	0	1	0	0	0	0	1	1
71	49.92	.871	0	0	1	0	0	0	1	1
72	50.62	.884	0	1	1	0	0	0	1	1
73	51.33	.896	0	0	0	1	0	0	1	1
74	52.03	.908	0	1	0	1	0	0	1	1
75	52.73	.920	1	1	0	1	0	0	1	1
76	53.44	.933	1	0	1	1	0	0	1	1
77	54.14	.945	1	1	1	1	0	0	1	1
78	54.84	.957	1	0	0	0	1	0	1	1
79	55.55	.969	1	1	0	0	1	0	1	1
80	56.25	.982	1	0	1	0	1	0	1	1
81	56.95	.994	0	1	1	0	1	0	1	1
82	57.66	1.006	0	0	0	1	1	0	1	1
83	58.36	1.019	0	1	0	1	1	0	1	1
84	59.06	1.031	1	1	0	1	1	0	1	1
85	59.77	1.043	1	0	1	1	1	0	1	1
86	60.47	1.055	0	1	1	1	1	0	1	1
87	61.17	1.068	0	0	0	0	0	1	1	1
88	61.87	1.080	0	1	0	0	0	1	1	1
89	62.58	1.092	1	1	0	0	0	1	1	1
90	63.28	1.104	0	0	1	0	0	1	1	1
91	63.98	1.117	0	1	1	0	0	1	1	1
92	64.69	1.129	1	1	1	0	0	1	1	1
93	65.39	1.141	0	0	0	1	0	1	1	1
94	66.09	1.154	0	1	0	1	0	1	1	1
95	66.80	1.166	1	1	0	1	0	1	1	1
96	67.50	1.178	0	0	1	1	0	1	1	1
97	68.20	1.190	1	0	1	1	0	1	1	1
98	68.91	1.203	1	1	1	1	0	1	1	1
99	69.61	1.215	0	0	0	0	1	1	1	1
100	70.31	1.227	1	0	0	0	1	1	1	1
101	71.02	1.239	0	1	0	0	1	1	1	1
102	71.72	1.252	1	1	0	0	1	1	1	1
103	72.42	1.264	0	0	1	0	1	1	1	1
104	73.12	1.276	1	0	1	0	1	1	1	1
105	73.83	1.289	0	1	1	0	1	1	1	1
106	74.53	1.301	0	1	1	0	1	1	1	1
107	75.23	1.313	1	1	1	0	1	1	1	1
108	75.94	1.325	0	0	0	1	1	1	1	1
109	76.64	1.338	1	0	0	1	1	1	1	1
110	77.34	1.350	0	1	0	1	1	1	1	1
111	78.05	1.362	0	1	0	1	1	1	1	1
112	78.75	1.374	1	1	0	1	1	1	1	1
113	79.45	1.387	1	1	0	1	1	1	1	1
114	80.16	1.399	0	0	1	1	1	1	1	1
115	80.86	1.411	0	0	1	1	1	1	1	1
116	81.56	1.424	1	0	1	1	1	1	1	1
117	82.27	1.436	1	0	1	1	1	1	1	1
118	82.97	1.448	0	1	1	1	1	1	1	1
119	83.67	1.460	0	1	1	1	1	1	1	1
120	84.38	1.473	1	1	1	1	1	1	1	1
121	85.08	1.485	1	1	1	1	1	1	1	1
122	85.78	1.497	1	1	1	1	1	1	1	1
123	86.48	1.509	1	1	1	1	1	1	1	1
124	87.19	1.522	1	1	1	1	1	1	1	1
125	87.89	1.534	1	1	1	1	1	1	1	1
126	88.59	1.546	1	1	1	1	1	1	1	1
127	89.30	1.559	1	1	1	1	1	1	1	1

Courtesy National Semiconductor Corp.

Semiconductor MM5220BM sine look-up ROM.

Table 7-2. A Sine Look-Up Table With Angles in 1° Increments

Angle	Sine		Angle	Sine	
	Decimal	Binary		Decimal	Binary
0.00	.0000	00000000	45.00	.7071	01011010
1.00	.0175	00000010	46.00	.7193	01011100
2.00	.0349	00000100	47.00	.7313	01011101
3.00	.0523	00000110	48.00	.7431	01011111
4.00	.0698	00001000	49.00	.7547	01100000
5.00	.0872	00001011	50.00	.7660	01100010
6.00	.1045	00001101	51.00	.7771	01100011
7.00	.1219	00001111	52.00	.7880	01100100
8.00	.1392	00010001	53.00	.7986	01100110
9.00	.1564	00010100	54.00	.8090	01100111
10.00	.1736	00010110	55.00	.8191	01101000
11.00	.1908	00011000	56.00	.8290	01101010
12.00	.2079	00011010	57.00	.8387	01101011
13.00	.2250	00011100	58.00	.8480	01101100
14.00	.2419	00011110	59.00	.8572	01101101
15.00	.2588	00100001	60.00	.8660	01101110
16.00	.2756	00100011	61.00	.8746	01101111
17.00	.2924	00100101	62.00	.8829	01110001
18.00	.3090	00100111	63.00	.8910	01110010
19.00	.3256	00101001	64.00	.8988	01110011
20.00	.3420	00101011	65.00	.9063	01110100
21.00	.3584	00101101	66.00	.9135	01110100
22.00	.3746	00101111	67.00	.9205	01110101
23.00	.3907	00110010	68.00	.9272	01110110
24.00	.4067	00110100	69.00	.9336	01110111
25.00	.4226	00110110	70.00	.9397	01111000
26.00	.4384	00111000	71.00	.9455	01111001
27.00	.4540	00111010	72.00	.9511	01111001
28.00	.4695	00111100	73.00	.9563	01111010
29.00	.4848	00111110	74.00	.9613	01111011
30.00	.5000	01000000	75.00	.9659	01111011
31.00	.5150	01000001	76.00	.9703	01111100
32.00	.5299	01000011	77.00	.9744	01111100
33.00	.5446	01000101	78.00	.9781	01111101
34.00	.5592	01000111	79.00	.9816	01111101
35.00	.5736	01001001	80.00	.9848	01111110
36.00	.5878	01001011	81.00	.9877	01111110
37.00	.6018	01001101	82.00	.9903	01111110
38.00	.6157	01001110	83.00	.9926	01111111
39.00	.6293	01010000	84.00	.9945	01111111
40.00	.6428	01010010	85.00	.9962	01111111
41.00	.6561	01010011	86.00	.9976	01111111
42.00	.6691	01010101	87.00	.9986	01111111
43.00	.6820	01010111	88.00	.9994	01111111
44.00	.6947	01011000	89.00	.9998	01111111
45.00	.7071	01011010	90.00	1.0000	01111111

register pair H into one of the 8080 general-purpose registers. What must be stored in the memory location addressed by the base address of the look-up table? The sine of 0° must be stored in memory at this address. Since an angle of 0° means that the A register will contain 0 when the SINANG subroutine is called, register pair B will contain 0 when it is added to the base address in register pair H. Therefore, the sine of 0° must be stored in memory at the base address of the look-up table.

How many memory locations must be used to store the sine look-up table? The look-up table will require 90_{10} memory locations, since the SINANG subroutine can be used to determine the sine of any angle between 0° and 90°. Of course, the SINANG subroutine can only be called with an integer value (between 0 and 90_{10}) in the A register. If the base address of the look-up table is 004 000 (0400), what will be the last memory address used by the look-up table? The last address will be 004 131 (0459). This means that 132 (5A) memory locations are used by the look-up table. What value will be stored in memory location 004 131 (0459)? The sine of 90°.

One nice feature of the SINANG subroutine, and a characteristic of using look-up tables in general, is the fact that it takes just as long to find one value in the table as it does to find another. This means that it takes the same amount of time for the 8080 to determine the sine of 32° as it does to determine the sine of 78°.

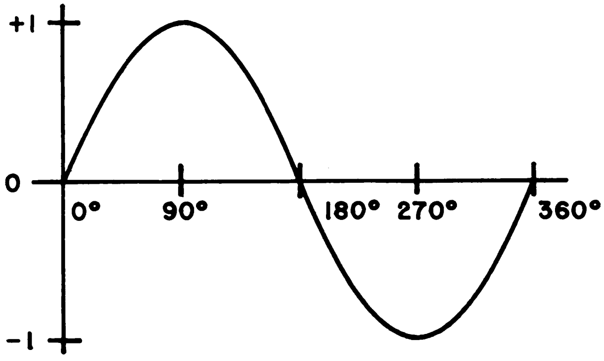


Fig. 7-2. The sine of all angles between 0° and 360°.

Suppose the sine of 131° or 222° must be determined. If the binary equivalents for these angles are stored in the A register when the SINANG subroutine is called, and we had not taken the precaution of comparing the content of the A register to 133, the 8080 would generate a table address that is greater than the final address of the look-up table. This is because the table only contains the sine of the angles between 0° and 90°.

The sine of all angles between 0° and 360° can be graphed, as shown in Fig. 7-2. From this graph, it is easy to see that the sine of all angles between 0° and 180° is positive and that the sine of all angles greater than 180° and less than 360° is negative. Therefore,

$$\begin{aligned} 0^\circ < X < 180^\circ&, \text{ sine is positive} \\ 180^\circ < X < 360^\circ&, \text{ sine is negative} \end{aligned}$$

As you can see in Fig. 7-2, the sine of 91° is the same as the sine of 89° and the sine of 181° is the same as the sine of 1°, irrespective of the sign of the sine. Therefore, it can be concluded that for angle X

$$\begin{aligned} 0^\circ \leqq X \leqq 90^\circ&, \text{ take sine}(X) \\ 90^\circ \leqq X \leqq 180^\circ&, \text{ take sine}(180^\circ - X) \\ &\quad \text{or sine}(90^\circ - (X - 90^\circ)) \end{aligned}$$

For example,

$$\begin{aligned} \text{sine}(170^\circ) &= \text{sine}(180^\circ - 170^\circ) \\ \text{sine}(170^\circ) &= \text{sine}(10^\circ) \end{aligned}$$

or

$$\begin{aligned} \text{sine}(170^\circ) &= \text{sine}(90^\circ = (170^\circ = 90^\circ)) \\ \text{sine}(170^\circ) &= \text{sine}(90^\circ - 80^\circ) \\ \text{sine}(170^\circ) &= \text{sine}(10^\circ) \end{aligned}$$

For the same angle X, the following can also be stated,

$$\begin{aligned} 180^\circ \leqq X \leqq 270^\circ&, \text{ take sine}(X - 180^\circ) \\ 270^\circ \leqq X \leqq 360^\circ&, \text{ take sine}(360^\circ - X) \text{ or} \\ &\quad \text{take sine}(90^\circ - (X - 270^\circ)) \end{aligned}$$

For example,

$$\begin{aligned} \text{sine}(190^\circ) &= \text{sine}(190^\circ - 180^\circ) \\ \text{sine}(190^\circ) &= \text{sine}(10^\circ) \end{aligned}$$

For an angle between 270° and 360°, such as 290°,

$$\begin{aligned} \text{sine}(290^\circ) &= \text{sine}(90^\circ - (290^\circ - 270^\circ)) \\ \text{sine}(290^\circ) &= \text{sine}(90^\circ - 20^\circ) \\ \text{sine}(290^\circ) &= \text{sine}(70^\circ) \end{aligned}$$

Of course, *the sign of the sine is negative for angles greater than 180° and less than 360°*. Based on these results, a flowchart can be prepared that illustrates the calculations required to reduce the sine of any angle between 0° and 360° to the sine of an angle between 0° and 90° (Fig. 7-3).

The subroutine listed in Example 7-2 can be used to determine the sine of any angle between 0° and 360°. The subroutine determines the seven-bit sine of the angle and also the sign of the sine. When the SINANG subroutine is called, the integer binary angle must be contained in register pair B (the B register contains the MSBY and the C register contains the LSBY). The angle is an unsigned number between 0° and 360° (000 000 through 001 150; 0000 through 0168).

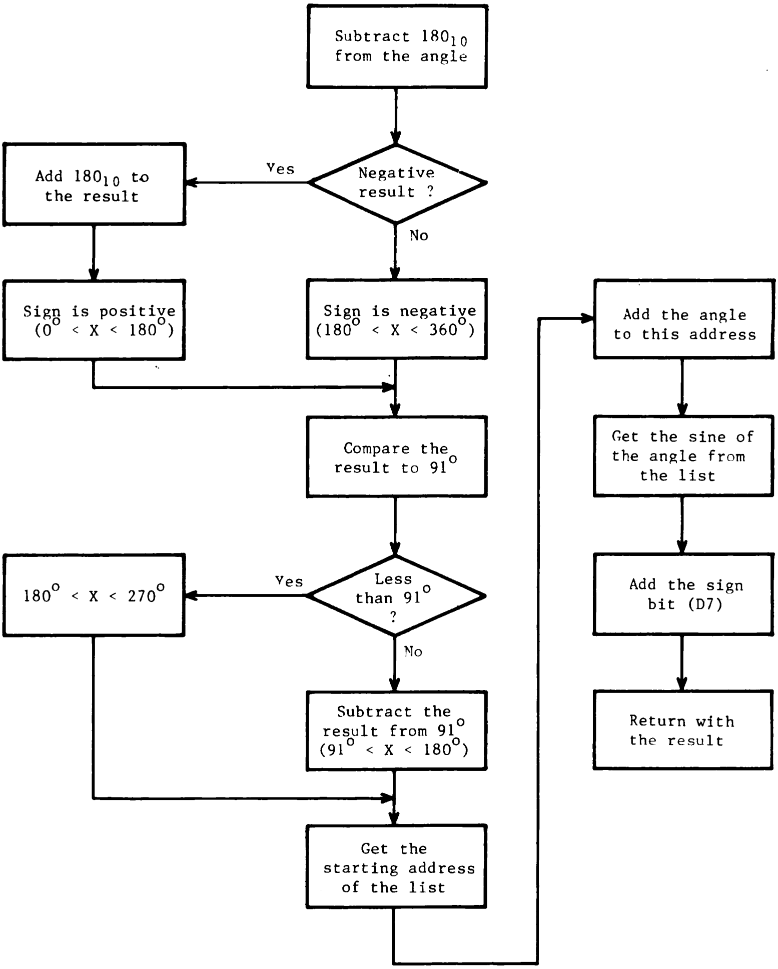

Subtract 180_{10} from the angle
Negative result ?
Yes
No
Add 180_{10} to the result
Sign is positive ($0^\circ < X < 180^\circ$)
Sign is negative ($180^\circ < X < 360^\circ$)
Compare the result to 91°
Less than 91° ?
Yes
No
$180^\circ < X < 270^\circ$
Subtract the result from 91° ($91^\circ < X < 180^\circ$)
Get the starting address of the list
Add the angle to this address
Get the sine of the angle from the list
Add the sign bit (D7)
Return with the result

Fig. 7-3. Flowchart for the angle-to-sine conversion subroutine.

The first two instructions in the SINANG subroutine save register pairs D and H on the stack. The sign of the sine is then determined by "subtracting" 180° from the angle contained in register pair B. Rather than subtract 180° from the angle, the 2s complement of

Example 7-2: Determining the Sine of an Angle Between 0° and 360° Using a Look-Up Table

```
/THIS SUBROUTINE CALCULATES THE SINE OF THE BINARY ANGLE
/CONTAINED IN REGISTER PAIR B (0 - 360 DEGREES). THE
/SIGNED SINE IS RETURNED IN THE A REGISTER.

SINANG, PUSHD   /SAVE  REGISTER PAIR D
        PUSHH   /AND REGISTER PAIR H.
        LXIH    /LOAD REGISTER PAIR H WITH THE 2'S
        114     /COMPLEMENT OF 180 (THIS IS A
        377     /NEGATIVE 180).
        DADB    /ADD THE ANGLE TO —180.
        JNC     /IF THERE IS NO CARRY, THE ANGLE
        A0180   /IS BETWEEN 0 AND 179 DEGREES.
        0
        MVIB    /THE ANGLE IS EQUAL TO OR GREATER THAN 180,
        200     /SO THE SIGN OF THE SINE IS MINUS.
        JMP     /THE B REGISTER STORES THE SIGN.
        NXTSUB  /THE ANGLE + (—180) IS STORED
        0       /IN REGISTER PAIR H.
A0180,  PUSHB   /THE SIGN IS +, GET THE ORIGINAL
        POPH    /ANGLE (0-180) INTO H AND L.
NXTSUB, PUSHH   /SAVE THE ANGLE.
        LXID    /LOAD REGISTER PAIR D WITH
        246     /A —90.
        377
        DADD    /ADD THIS TO THE ANGLE IN H AND L.
        JNC     /IF THERE IS NO CARRY, THE ORIGINAL
        OKASIS  /IS BETWEEN 0 AND 90 OR 180 AND 270.
        0
        POPD    /POP D AND E IF THE ANGLE IS BETWEEN
        MVIA    /90-180 OR 270-360.
        132     /AND SUBTRACT THE RESULT OF
        SUBL    /THE ANGLE + (—180) FROM 90.
        MOVLA   /SAVE THE RESULT BACK IN L.
        JMP     /THEN FIND THE APPROPRIATE
        CALCIT  /ENTRY IN THE LOOK-UP TABLE.
        0
OKASIS, POPH    /GET THE ANGLE BACK IN H AND L.
CALCIT, LXID    /LOAD REGISTER PAIR D WITH THE
        SINTAB  /BASE ADDRESS OF THE SINE
        0       /LOOK-UP TABLE.
        DADD    /ADD THIS ADDRESS TO H AND L.
        MOVAM   /MOVE THE SINE OF THE ANGLE INTO A.
        ADDB    /ADD THE SIGN (000 OR 200).
        POPH    /POP REGISTER PAIR H AND REGISTER
        POPD    /PAIR D OFF OF THE STACK.
        CPI     /IS THE RESULT NEGATIVE ZERO?
        200
        RNZ     /NO, THEN RETURN.
        XRAA    /YES, THEN SET A TO ZERO.
        RET
SINTAB, 000     /THE SINE OF 0 DEGREES IS ZERO.
         •      /THE REMAINDER OF THE LOOK-UP
         •      /TABLE IS STORED HERE.
         •
```

180° is *added* to the angle. Therefore, register pair H is loaded with 377 114 (FF4C), and the DADB instruction performs the required addition. If the angle in register pair B is between 0° and 179°, the sign of the sine is positive, so the JNC to A0180 is executed. If the angle is greater than 180°, the sign of the sine is negative, so the 8080 does not jump to A0180. Instead, bit D_7 of the B register is set to a logic 1 and the 8080 jumps to NXTSUB.

If the sign of the sine is positive, the JNC to A0180 is executed. If the angle is between 0° and 180°, bit D_7 of the B register must be set to a logic 0. However, bit D_7 of the B register is *already* a logic 0 if the angle is between 0° and 180°. Therefore, the 8080 does not have to perform any special operations to set bit D_7 of the B register to a logic 0. Starting at A0180, the 8080 pushes the angle in register pair B onto the stack and then pops the angle off of the stack into register pair H.

If the angle is equal to or greater than 181°, the 8080 jumps to NXTSUB, so that the result of the "subtraction" (2s complement addition) is left in register pair H. If register pair B originally contained an angle of 181°, register pair H now contains an angle of 1°. If register pair B contained 271°, register pair H now contains 91°. *Regardless of the size of the angle originally (0° through 360°), register pair H now contains an angle between 0° and 180°, and bit D_7 of the B register contains the sign of the sine to be found in the look-up table.*

At NXTSUB, this angle is saved on the stack and register pair D is loaded with the 2s complement of 90°. This value is added to the angle in register pair H (between 0° and 180°). If the carry is a logic 0 as a result of this addition, the *original* angle is between 0° and 90° *or* between 180° and 270°. As an example, note that the only difference between the sine of 30° and the sine of 210° is the *sign* of the sine. The magnitude of the sine (0.5000) is the same for both angles. Therefore, by jumping to OKASIS, the 8080 can add the content of register pair H to the base address of the table. This means that the original angle is between 0° or 90°, or the content of register pair H was obtained by adding −180° to an angle between 180° and 270°.

If the carry is a logic 1 after −90 is added to the content of register pair H, the original angle must have been between either 90° and 180° or between 270° and 360°. Of course, by adding −180 to the original angle, the angles between 270° and 360° are converted to angles between 90° and 180°. Therefore, if the carry is a logic 1, the JNC to OKASIS is not executed. The POPD instruction pops the angle off of the stack into register pair H and the A register is loaded with 90_{10}. The result of adding −90 to the angle must now be subtracted from 90°. After the A register is loaded with 132 (hex 5A, decimal 90), the content of the L register is subtracted from the A register. We do not have to perform a 16-bit subtraction here, because any angle between 0° and 90° will be contained in a single eight-bit register.

If the 8080 jumps to OKASIS, the angle on the stack is popped off of the stack into register pair H. If the content of the L register was just subtracted from the 90_{10} contained in the A register, the 8080 jumps over this instruction. In either case, register pair D is loaded with the base address of the sine look-up table at CALCIT. This base address is then added to the angle (between 0° and 90°) contained in register pair H. After the DADD instruction is executed, the 8080 moves the seven-bit sine value from the look-up table to the A register, and the sign bit contained in the B register is added to this value. Register pairs H and D are then popped off of the stack. The 8080 then compares the signed sine in the A register to the immediate data byte 200 (80). This value is equal to a signed sine of negative 0. If the content of the A register is not equal to this, the 8080 returns. If the content of the A register is equal to a negative 0, the 8080 sets the content of the A register to a positive 0 before returning from the subroutine. If you are not worried about positive and negative 0s, the CPI, RNZ, and XRAA instructions can be eliminated. Without these instructions, the 8080 will determine that the signed sine of 180° is a negative 0.

USING A MORE ACCURATE SINE LOOK-UP TABLE

Now that the SINANG subroutine has been discussed, what is the accuracy of the sine of the angle? The sine of the angle is represented by a seven-bit number (the MSB of the eight-bit value contains the sign of the sine), so the sine of the angle is only accurate to 0.78% (1 part in 128). Suppose more accuracy is needed, such as 0.003%. How can the sine of an angle be determined with this much accuracy?

The simplest method would be to change the sine look-up table (SINTAB) from seven-bit to 15-bit values. By modifying our BASIC program, we had our 8080-based microcomputer produce the 16-bit sine look-up table listed in Table 7-3. The MSB of these values is 0, so that the sign of the sine can be added to the value obtained from the look-up table.

Of course, one of the first problems that must be solved is how to store these 16-bit values in memory. It does not matter whether the MSBY of each value is stored first or last in memory, as long as the SINANG subroutine is written to read both the MSBY and the LSBY at the appropriate time. We will assume that the LSBY of a value is stored in memory, followed by the MSBY in the next

Table 7-3. A 16-Bit (15-Bit + Sign) Sine Look-Up Table Accurate to 0.003%

Angle	Sine		Angle	Sine	
	Decimal	Binary		Decimal	Binary
0.00	.0000	0000000000000000	45.00	.7071	0101101010000010
1.00	.0175	0000001000111011	46.00	.7193	0101110000010011
2.00	.0349	0000010001110111	47.00	.7313	0101110110011100
3.00	.0523	0000011010110010	48.00	.7431	0101111100011111
4.00	.0698	0000100011101101	49.00	.7547	0110000010011010
5.00	.0872	0000101100100111	50.00	.7660	0110001000001101
6.00	.1045	0000110101100001	51.00	.7771	0110001101111001
7.00	.1219	0000111110011001	52.00	.7880	0110010011011101
8.00	.1392	0001000111010000	53.00	.7986	0110011000111001
9.00	.1564	0001010000000101	54.00	.8090	0110011110001101
10.00	.1736	0001011000111010	55.00	.8191	0110100011011001
11.00	.1908	0001100001101100	56.00	.8290	0110101000011101
12.00	.2079	0001101010011100	57.00	.8387	0110101101011001
13.00	.2250	0001110011001011	58.00	.8480	0110110010001100
14.00	.2419	0001111011110111	59.00	.8572	0110110110110111
15.00	.2588	0010000100100000	60.00	.8660	0110111011011001
16.00	.2756	0010001101001000	61.00	.8746	0110111111110011
17.00	.2924	0010010101101100	62.00	.8829	0111000100000100
18.00	.3090	0010011110001101	63.00	.8910	0111001000001100
19.00	.3256	0010100110101100	64.00	.8988	0111001100001011
20.00	.3420	0010101111000111	65.00	.9063	0111010000000001
21.00	.3584	0010110111011110	66.00	.9135	0111010011101111
22.00	.3746	0010111111110011	67.00	.9205	0111010111010010
23.00	.3907	0011001000000011	68.00	.9272	0111011010101101
24.00	.4067	0011010000001111	69.00	.9336	0111011101111111
25.00	.4226	0011011000011000	70.00	.9397	0111100001000111
26.00	.4384	0011100000011100	71.00	.9455	0111100100000110
27.00	.4540	0011101000011100	72.00	.9511	0111100110111100
28.00	.4695	0011110000010111	73.00	.9563	0111101001101000
29.00	.4848	0011111000001110	74.00	.9613	0111101100001010
30.00	.5000	0100000000000000	75.00	.9659	0111101110100011
31.00	.5150	0100000111101100	76.00	.9703	0111110000110010
32.00	.5299	0100001111010100	77.00	.9744	0111110010111000
33.00	.5446	0100010110110110	78.00	.9781	0111110100110011
34.00	.5592	0100011110010011	79.00	.9816	0111110110100101
35.00	.5736	0100100101101010	80.00	.9848	0111111000001110
36.00	.5878	0100101100111100	81.00	.9877	0111111001101100
37.00	.6018	0100110100001000	82.00	.9903	0111111011000000
38.00	.6157	0100111011001101	83.00	.9926	0111111100001011
39.00	.6293	0101000010001101	84.00	.9945	0111111101001100
40.00	.6428	0101001001000110	85.00	.9962	0111111110000011
41.00	.6561	0101001111111001	86.00	.9976	0111111110110000
42.00	.6691	0101010110100101	87.00	.9986	0111111111010010
43.00	.6820	0101011101001011	88.00	.9994	0111111111101011
44.00	.6947	0101100011101010	89.00	.9998	0111111111111010
45.00	.7071	0101101010000010	90.00	1.0000	0111111111111111

consecutive memory location at a higher address. For the angles 1° and 2°, the 16-bit values are stored in the look-up table as follows:

Angle		Octal *Memory Address*	Data	Hex *Memory Address*
1°	LSBY	015 023	00111011	0D13
	MSBY	015 024	00000010	0D14
2°	LSBY	015 025	01110111	0D15
	MSBY	015 026	00000100	0D16

The address of 015 023 (0D13) was arbitrarily chosen. As always, once a data format for the table is chosen, all other nodes must be stored in the same format. Therefore, the LSBYs of all the remaining values must be stored in memory first, at a lower memory address, followed immediately by the MSBY in the next higher consecutive memory location.

How many memory locations will be required to store this new sine look-up table? The answer is $90_{10} \times 2_{10}$, or 180_{10}. What modifications have to be made to the SINANG subroutine (Example 7-2)? Starting at CALCAD, the subroutine would have to be changed as shown in Example 7-3.

Example 7-3: Modifying the SINANG Subroutine (Example 7-2) To Operate With 16-Bit Sine Values

```
        •
        •
CALCAD, LXID    /THE ANGLE IS IN REGISTER PAIR H.
        SINTAB  /LOAD REGISTER PAIR D WITH THE
        0       /BASE ADDRESS OF THE SINE LOOK-UP TABLE.
        DADH    /MULTIPLY THE ANGLE BY 2.
        DADD    /ADD THE BASE ADDRESS TO (ANGLE*2).
        MOVAM   /MOVE THE LSBY OF THE SINE TO THE
        MOVCA   /A REGISTER AND THEN TO THE C REGISTER.
        INXH    /INCREMENT THE MEMORY ADDRESS.
        ORAM    /OR THE MSBY AND THE LSBY.
        JZ      /THE SINE IS ZERO, SO DON'T
        NEGZER  /ADD THE SIGN BIT TO THE SINE.
        0
        MOVAM   /GET THE MSBY OF THE SINE INTO A.
        ADDB    /ADD THE SIGN TO THE SEVEN-BIT MSBY.
NEGZER, MOVBA   /SAVE THE MSBY OF THE SINE IN B.
        POPH    /POP REGISTER PAIR H AND REGISTER
        POPD    /PAIR D OFF OF THE STACK.
        RET
```

The first portion of the subroutine will remain exactly the same as the previous example. The sign of the sine will still be calculated in the same manner. Also, all angles between 0° and 360° will be mathematically reduced to angles between 0° and 90°.

Starting at CALCAD, the base address of the look-up table is loaded into register pair D. The DADH instruction multiplies the angle between 0° and 90° by 2, because each node in the look-up table requires two memory locations for storage. This means that the sine of 40° is no longer 40_{10} memory locations from the beginning of the look-up table, but rather 80_{10} memory locations from the beginning of the look-up table. The base address of the look-up table, which is contained in register pair D, is then added to the "multiplied" angle in register pair H. The LSBY of the sine of the angle is then moved to the A and C registers. The memory address in register pair H is incremented, and the LSBY in the A register is ORed with the MSBY in memory. If the sine of the angle is 0, the 8080 jumps to NEGZER, otherwise the MSBY of the sine is moved to the A register where the sign bit is added to it. At NEGZER, the MSBY of the sine, with the sign bit, is saved in the B register. Register pairs H and D are then popped off of the stack and the 8080 returns from the SINANG subroutine. Register pair B contains the sine of the angle.

Like the previous software example, the CALCAD section of the subroutine contains instructions so that a sine of negative 0 is not produced. By ORing the LSBY and MSBY of the sine, the 8080 determines if the sine of the angle is 0. If it is, the 8080 jumps to NEGZER, so that the 0 contained in the A register is saved in the B register. This means that the sign bit (D_7 of the B register) is 0. If the sine of the angle is nonzero, the 8080 does not execute the JZ to NEGZER. Instead, it adds the sign bit to the MSBY of the sine value.

One problem that has not been discussed is what will happen in the SINANG subroutine if an angle larger than 360° is contained in register pair B when the subroutine is called. There is no way of predicting what will be contained in the A register (or register pair B, depending on the accuracy of the sine look-up table) when the 8080 returns. Of course, the reason for this is that the 8080 will generate a memory address that is greater than the highest address used by the sine look-up table. The simplest way to avoid this problem would be to check the content of register pair B before any calculations are performed. If the content of register pair B is 360° or less, the remainder of the subroutine can be executed. If the content of register pair B is greater than 360°, then 360° can be subtracted from this angle and the result compared to 361°. If the result is equal to or greater than 361°, then the 8080 should continue the subtract-and-compare process until the angle in register pair B is reduced to an angle between 0° and 360°. When the angle has finally been "reduced" to an angle within these bounds, the remainder of the SINANG subroutine can be executed. For some angles, this means that 360° will have to be subtracted from the angle more than once.

A PAPER-TAPE LETTERER PROGRAM

If you have ever used paper tape for storing programs or data, you know that it is sometimes difficult to write a descriptive title at the beginning of the tape. Most pens do not write well on the oiled paper tape and many inks blur. To solve this problem, it would be convenient if there were a program that could be used to punch a message on the paper tape when the required keys on the teletypewriter or crt are pressed. For instance, if the T, E, S, and T keys are pressed, the alphanumeric characters TEST are punched on the paper tape in the form of 5 × 7 dot-matrix characters. The program that does this is called a *paper-tape letterer program,* and it is another example of a program that uses a look-up table.

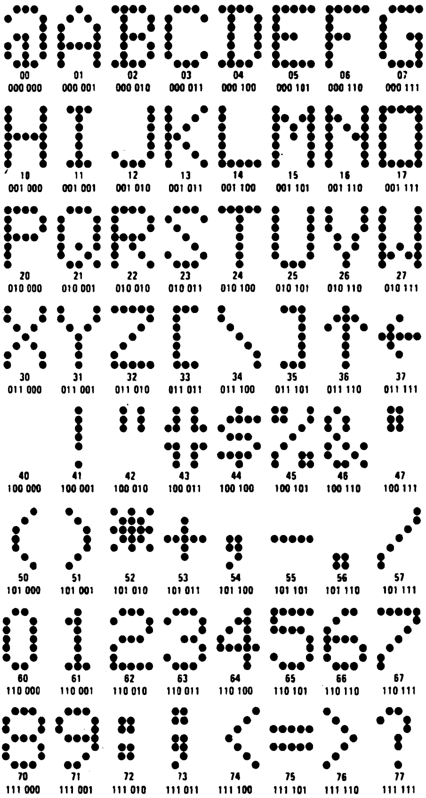
The 5 × 7 dot matrices that should be produced by the computer when the keys on the teletypewriter or crt are pressed are shown in Fig. 7-4. Suppose that the N key on the teletypewriter or crt is pressed. What values will have to be punched on the paper tape? The 5 × 7 dot matrix for this character is shown in Fig. 7-5.

The first value that would be punched on the paper tape would be 177 (7F). Even though most teletypewriters can punch eight bits of data in parallel, we have only used seven of the eight "channels." Therefore, the eighth and most-significant channel will not be used, so it will always be a logic 0 (no hole punched). Then the values 002, 004, 010, and 177 (02, 04, 08, and 7F) would be punched. The paper tape is punched from left to right, because this is the way that the paper tape is read.

We have assumed from Figs. 7-4 and 7-5 that a dot represents a hole punched in the paper tape. Where are the values 177, 002, 004, 010, and 177 (7F, 02, 04, 08, and 7F) stored? These values are stored in a look-up table, along with 63 other 5 × 7 dot matrices. Since each dot matrix is five data values "long," the entire look-up table will require 320_{10} memory locations for storage.

If required, the look-up table could be composed of variable-length entries. This means that each "character" does not have to be represented by a 5 × 7 dot matrix. The advantage of using variable-length entries is that not all characters require a 5 × 7 dot matrix to be properly represented. For instance, to store an exclamation mark as a 5 × 7 dot matrix, two 0s, a 137 (5F), followed by two additional 0s, would have to be stored in the look-up table.

Rather than use a 5 × 7 dot matrix, we might just store a 137 (5F) in memory, followed by a termination character. This means that


00 01 02 03 04 05 06 07
000 000 000 001 000 010 000 011 000 100 000 101 000 110 000 111
10 11 12 13 14 15 16 17
001 000 001 001 001 010 001 011 001 100 001 101 001 110 001 111
20 21 22 23 24 25 26 27
010 000 010 001 010 010 010 011 010 100 010 101 010 110 010 111
30 31 32 33 34 35 36 37
011 000 011 001 011 010 011 011 011 100 011 101 011 110 011 111
40 41 42 43 44 45 46 47
100 000 100 001 100 010 100 011 100 100 100 101 100 110 100 111
50 51 52 53 54 55 56 57
101 000 101 001 101 010 101 011 101 100 101 101 101 110 101 111
60 61 62 63 64 65 66 67
110 000 110 001 110 010 110 011 110 100 110 101 110 110 110 111
70 71 72 73 74 75 76 77
111 000 111 001 111 010 111 011 111 100 111 101 111 110 111 111

Courtesy National Semiconductor Corp.

Fig. 7-4. The 5 × 7 dot matrices stored in a look-up table.

only two memory locations will be required rather than five. Another storage method that could be used would be to store the number of memory locations used per character in one memory location, followed by the memory location(s) used to store the character. Using either the termination or "node-count" method, three memory locations can be saved using this variable-length entry for the exclamation-mark character.

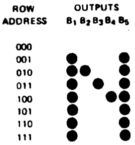

ROW ADDRESS
OUTPUTS $B_1 B_2 B_3 B_4 B_5$
000
001
010
011
100
101
110
111

Fig. 7-5. The 5 × 7 dot matrix for the character "N."

Courtesy National Semiconductor Corp.

Unfortunately, most of the 64 characters in the ASCII character set require five memory locations to store the dot matrix. Therefore, if a variable-length table is created, many of the dot matrices will require six memory locations for storage—five for the dot matrix and one for the "node count" or the matrix terminator. We have calculated that a total of 349 memory locations will be required for storing the variable-length table. Only 320 memory locations are required to store the 64 dot matrices, if *each* one uses five memory locations. For this reason, the program that will be used to punch the alphanumeric characters on paper tape will access a look-up table that contains 64 nodes, where each node requires five memory locations for storage. This program, which is listed in Example 7-4, inputs a character from the teletypewriter keyboard, looks up the proper 5 × 7 dot matrix in the look-up table, and punches the dot matrix on paper tape.

At the beginning of this program (Example 7-4), the stack pointer is loaded with a R/W memory address. The TTYI subroutine is then called. The 8080 will return from this subroutine with the eight-bit ASCII value for for the key that is pressed in the A register. *The character will not be printed (echoed) on the printer.*

Why did we use the TTYI subroutine in Example 7-4, rather than the TTYIN subroutine? What are the differences between the two subroutines? The TTYI subroutine inputs a character from the keyboard, but the character is *not* printed on the printer. The TTYIN subroutine also inputs an ASCII value from the keyboard, but this value is also transmitted back to the printer, so that the character is printed. Quite often, the printer and punch mechanisms of a teletypewriter are mechanically linked together. Since we want to punch a special sequence of dots on the paper tape, the *ASCII value* for the key that is pressed *must not* be transmitted to the printer. If it is, the ASCII value for the key will be punched on the paper tape, along with the appropriate 5 × 7 dot matrix. Therefore, as keys are pressed on the keyboard the message is not printed on the printer.

When the 8080 returns from the TTYI subroutine, the eight-bit ASCII value is in the A register. The parity bit is then set to 0 by the ANI instruction. The 8080 now has to determine if the ASCII value received represents a character whose dot matrix is stored in the look-up table. For instance, the characters RETURN (CR; 015, 0D), LINE FEED (LF; 012, 0A), and BELL (007, 07) cannot be represented by a dot matrix. You can see this by examining Fig. 7-4. There are no dot matrices in this figure for these three ASCII characters. Only the characters between SPACE (040, 20) and UNDERLINE (137, 5F) can be represented by a 5 × 7 dot matrix. Therefore, after the 8080 returns from the TTYI subroutine, the

Example 7-4: A Paper-Tape Letterer Program

```
LETTER,  LXISP   /LOAD THE STACK POINTER WITH A
         STACK   /R/W MEMORY ADDRESS.
         0
IGNOR,   CALL    /GET A KEYBOARD CHARACTER BUT
         TTYI    /DO NOT PRINT THE CHARACTER
         0       /(NO ECHO).
         ANI     /SET THE PARITY BIT TO ZERO.
         177
         CPI     /IS THE CHARACTER LESS THAN
         040     /040 (20), THE ASCII VALUE FOR
         JC      /A SPACE? YES, THEN IGNORE
         IGNOR   /THE CHARACTER.
         0
         CPI     /IS THE CHARACTER GREATER THAN
         140     /AN UNDERLINE?
         JNC     /YES, THEN IGNORE IT ALSO.
         IGNOR
         0
         SUI     /IT WAS A VALID CHARACTER, SO
         040     /SUBTRACT 040 (20) FROM THE ASCII VALUE.
         MOVLA   /ALL VALUES ARE NOW 000 - 077 (00 - 7F).
         MVIH    /SAVE THE VALUE IN L, SET H TO 000.
         000
         PUSHH   /SAVE THE VALUE ON THE STACK.
         POPD    /THEN POP IT OFF INTO D.
         DADH    /MULTIPLY THE VALUE BY 2.
         DADH    /NOW BY 2 AGAIN (TOTAL=X4).
         DADD    /ADD THE ORIGINAL NUMBER (TOTAL=X5).
         LXID    /NOW LOAD REGISTER PAIR D WITH THE
         TABLE   /STARTING ADDRESS FOR THE DOT
         0       /MATRIX LOOK-UP TABLE.
         DADD    /ADD THE ADDRESS TO THE X5 ASCII VALUE.
FIRST,   MVIC    /THERE ARE FIVE MEMORY LOCATIONS
         005     /PER CHARACTER DOT MATRIX.
NXTCHR,  MOVAM   /GET AN EIGHT-BIT CHARACTER.
         CALL    /PUNCH IT ON THE PAPER TAPE PUNCH
         TTYOUT  /THAT IS MECHANICALLY LINKED
         0       /TO THE TELETYPEWRITER'S PRINTER.
         INXH    /INCREMENT THE MEMORY ADDRESS.
         DCRC    /DECREMENT THE COUNT.
         JNZ     /THE COUNT IS NONZERO,
         NXTCHR  /SO GET ANOTHER CHARACTER FROM MEMORY
         0       /AND PUNCH IT OUT.
         XRAA    /PUNCHED ALL FIVE COLUMNS, SO NOW
         CALL    /PUNCH A BLANK COLUMN TO SEPARATE
         TTYOUT  /THE CHARACTERS IN THE MESSAGE.
         0
         JMP     /GET ANOTHER CHARACTER FROM THE
         IGNOR   /KEYBOARD.
         0
TTYI,    IN      /INPUT THE UART'S STATUS.
         001
         ANI     /SAVE ONLY THE RECEIVER'S STATUS.
         001
```

```
        JZ       /IF A=001, A KEY IS PRESSED.
        TTYI     /IF A=000, NO KEY IS PRESSED.
        0        /IF NO KEY IS PRESSED, KEEP WAITING.
        IN       /A KEY IS PRESSED, SO INPUT THE
        000      /CHARACTER'S ASCII CODE.
        RET      /AND RETURN WITH IT IN THE A REGISTER.

TTYOUT, MOVBA    /SAVE THE CHARACTER IN B.
TTYO,   IN       /INPUT THE UART'S STATUS WORD.
        001
        ANI      /SAVE ONLY THE TRANSMITTER'S FLAG.
        004      /IF A=004, THE TRANSMITTER (PRINTER) IS READY.
        JZ       /IF A=000, THE TRANSMITTER (PRINTER) IS BUSY.
        TTYO     /SO KEEP WAITING FOR THE TRANSMITTER
        0        /(PRINTER) TO FINISH, BEFORE THE
        MOVAB    /CONTENT OF THE A REGISTER CAN BE PRINTED.
        OUT      /AFTER THE CHARACTER IS MOVED FROM
        000      /B TO A, OUTPUT IT TO THE UART.
        RET      /RETURN WITH THE CHARACTER STILL IN A.

TABLE,  000      /THIS IS THE SPACE CHARACTER
        000
        000
        000
        000
EXCLAM, 000      /EXCLAMATION MARK
        000
        137
        000
        000
         •
         •
```

content of the A register is ANDed with 177 (7F), so that the parity bit is set to 0. The result is compared to 040 (20). If the ASCII value for the key that is pressed is less than 040 (20), the JC to IGNOR is executed. Likewise, if the ASCII value for the key that is pressed is equal to or greater than 140 (60), the JNC to IGNOR is executed. Only if the ASCII value received is between 040 and 137 (20 and 5F), will the SUI 040 instruction be executed. This instruction causes 040 (20) to be subtracted from any of the "valid" ASCII values so that the A register contains a value between 0 and 077 (0 and 3F). This number now has to be used by the 8080 to look up the proper 5 × 7 dot matrix in the table.

Suppose that you press the space bar on the teletypewriter. The 8080 receives the ASCII value 040 (20). Since this value is between the limits of 040 and 137 (20 and 5F), the 8080 does not jump back to IGNOR. Instead, it subtracts 040 (20) from the ASCII value in the A register. The result of this subtraction is 0 (040 − 040 or 20 − 20), which is saved in the L register. The H register is then cleared to 0 by the MVIH instruction. Register pair H now contains 0 as a result of the space bar being pressed on the teletypewriter keyboard. What will be contained in register pair H at this point in the program if the exclamation-mark key is pressed? The ASCII value for the exclamation mark is 041 (21), so register pair H would contain 000 001 (0001). Using these numbers, the 8080 has to be able to find the appropriate dot matrix in the look-up table.

You already know that each character is represented by a 5 × 7 dot matrix and that each dot matrix is stored in *five consecutive memory locations.* A portion of one possible dot-matrix look-up table is shown in Table 7-4. Both octal and hexadecimal addresses and data values are shown. Based on this table, the 8080 must generate the address 020 000 (1000) when the space bar is pressed, and the address 020 005 (1005) when the exclamation-mark key is pressed. The simplest method of doing this would be to multiply the result of the subtract-immediate instruction by 5, and then add this result to the base address of the look-up table. For the space bar, the result of the multiplication is 0 (000 000 × 005 = 000 000). When this number is added to the base address of the table, the result *is* the base address. For the exclamation-mark key, the subtraction result of 1 is multiplied by 5 and the result is 5, which is added to the base address. The result is the address 020 005 (1005). This is the memory address where the first portion of the 5 × 7 dot matrix for the exclamation mark must be stored in memory.

In Example 7-4, after the H register is loaded with 0, the content of register pair H is pushed onto the stack and popped off of the stack into register pair D. Register pairs H and D now contain

Table 7-4. The Organization of the Dot-Matrix Look-Up Table

Octal		Hexadecimal	
Address	Content	Address	Content
020 000	TABLE, 000	1000	TABLE, 00
020 001	000	1001	00
020 002	000	1002	00
020 003	000	1003	00
020 004	000	1004	00
020 005	EXCLAM, 000	1005	EXCLAM, 00
020 006	000	1006	00
020 007	137	1007	5F
020 010	000	1008	00
020 011	000	1009	00
020 012	QUOTE, 000	100A	QUOTE, 00
020 013	007	100B	07
020 014	000	100C	00
• •	•	•	•
• •	•	•	•
• •	•	•	•

the same 16-bit number. For the math that you already know, you know that multiplication by 5 is equal to multiplication by 4 followed by addition,

$$A \times 5 = A \times 4 + A$$

For this reason, two consecutive DADH instructions are executed after register pair H is popped off of the stack into register pair D. These two instructions cause the content of register pair H to be multiplied by 4. The addition of the original number, in register pair D, to the multplication result in register pair H, completes the multiplication by 5. The result of this multiplication is contained in register pair H. Register pair D is then loaded with the base address of the look-up table, and this address is added to the number in register pair H by the DADD instruction just before FIRST. Register pair H now contains the address for the first "character" in the proper 5 × 7 dot matrix for the key that was pressed on the teletypewriter. The 8080 now has to read five consecutive characters from memory, and punch them on the paper tape.

Since only five values are punched on paper tape for each character, a count of five is loaded into the C register at FIRST. The content of memory addressed by register pair H is then moved to the A register, and the TTYOUT subroutine is called so that the character is punched on the paper tape. The TTYOUT subroutine has been used previously to print characters on the printer. However, since the printer and punch mechanisms are mechanically linked together, the TTYOUT subroutine can be used to punch the content of the A register on paper tape. After the character is punched, the 8080 returns to the INXH instruction.

This instruction causes the memory address to be incremented, and then the count in the C register is decremented. If five characters have not been punched, the 8080 jumps back to NXTCHR. When the contents of all five memory locations have been punched on the paper tape, the 8080 clears the A register to 0, and this value is punched on the paper tape after the 5 × 7 dot matrix. This creates a space on the paper tape so that the dot matrices for each character are separated by a "space." Note that the ASCII value for SPACE (040, 20) *is not punched on the paper tape.* If this value is punched, a space will be printed on the printer, *and* the ASCII value for space will be punched on the paper tape. The 5 × 7 dot matrices will not be separated by a space. Remember, we are not concerned with what is printed on the printer. We are only concerned with the characters that are punched on the paper tape.

To ensure that you understand how the 8080 calculates the address for the proper dot matrices, assume that the Z key is pressed.

If the base address of the look-up table is 020 000 (1000), where is the dot matrix for the Z key stored?

Octal	Hex
132	5A
−040	−20
072	3A
× 5	×5
001 042	0122
+020 000	+1000
021 042	1122

The dot matrix for the Z key will be stored in memory from 021 042 through 021 046 (1122 through 1126). The *complete* paper-tape letterer look-up table is listed in Table 7-5.

Can you think of one method that can be used to simplify the program listed in Example 7-4? Do *not* change the organization of the look-up table. The beginning of the program can be modified, as shown in Example 7-5. In Example 7-5, the TTYI subroutine is called after the stack pointer is loaded with a R/W memory address. When the 8080 returns, bit D_7 of the A register is set to a 0 by the ANI instruction. The value 040 (20) is then subtracted from the content of the A register. By subtracting 040 (20) from the ASCII values, the *valid* characters, the characters between SPACE and UNDERLINE, now have values between 000 and 077 (00 and 3F). The CPI instruction then determines whether the character entered is valid. If the content of the A register is less than 100 (40), the

Example 7-5: Simplifying the Paper-Tape Letterer Program

```
LETTER, LXISP   /LOAD THE STACK POINTER.
        STACK
        0
IGNOR,  CALL    /GET A KEYBOARD CHARACTER BUT
        TTYI    /DO NOT PRINT THE CHARACTER
        0       /(NO ECHO).
        ANI     /SET THE PARITY BIT (D7) TO ZERO.
        177
        SUI     /SUBTRACT 040 (20) FROM ALL THE ASCII
        040     /CHARACTERS. VALID CHARACTERS HAVE
        CPI     /THE VALUES 000 - 077 (00 - 3F).
        100     /ANYTHING GREATER IS INVALID.
        JNC     /THE VALUE IS GREATER THAN 077,
        IGNOR   /SO IGNORE IT.
        0
        MOVLA   /SAVE THE VALUE IN L.
        MVIH    /SET H TO 000.
        000
        •
        •
```

Table 7-5. The Complete Paper-Tape Letterer Look-Up Table

TABLE,	000	PLUS,	010	SIX,	074	A,	174	L,	177	W,	177
	000		010		112		022		100		040
	000		076		111		021		100		020
	000		010		111		022		100		040
	000		010		060		174		100		177
EXCLAM,	000	COMMA,	130	SEVEN,	141	B,	101	M,	177	X,	143
	000		070		021		177		002		024
	137		000		011		111		014		010
	000		000		005		111		002		024
	000		000		003		066		177		143
QUOTE,	000	DASH,	010	EIGHT,	044	C,	076	N,	177	Y,	003
	007		010		111		101		002		004
	000		010		111		101		004		170
	007		010		111		101		010		004
	000		010		044		042		177		003
NSIGN,	014	PERID,	000	NINE,	006	D,	101	O,	177	Z,	141
	167		000		111		177		101		121
	000		140		111		101		101		111
	167		140		041		101		101		105
	014		000		036		042		177		103
DSIGN,	044	SLASH,	140	COLON,	000	E,	177	P,	177	RSB,	000
	052		020		000		111		011		177
	153		010		066		111		011		101
	052		004		066		101		011		101
	022		003		000		101		006		000
PCENT,	143	ZERO,	000	SCOLON,	000	F,	177	Q,	076	BSLSH,	003
	023		076		000		011		101		004
	010		101		133		011		121		010
	144		101		073		001		041		020
	143		076		000		001		136		140
AND,	044	ONE,	000	LAROW,	000	G,	076	R,	177	LSB,	000
	111		102		010		101		011		101
	126		177		024		101		031		101
	040		100		042		111		051		177
	120		000		101		171		106		000
APOS,	000	TWO,	162	ESIGN,	024	H,	177	S,	042	UPAROW,	004
	007		111		024		010		105		002
	007		111		024		010		111		177
	000		111		024		010		121		002
	000		106		024		177		042		004
RBRAC,	034	THREE,	042	RAROW,	101	I,	000	T,	001	BAROW,	004
	042		101		042		101		001		016
	101		111		024		177		177		021
	000		111		010		101		001		004
	000		066		000		000		001		004
LBRAC,	000	FOUR,	030	QMARK,	000	J,	040	U,	077		
	000		024		002		100		100		
	101		022		001		100		100		
	042		177		131		100		100		
	034		020		006		177		077		
STAR,	025	FIVE,	047	APPRX,	062	K,	177	V,	007		
	016		105		111		010		030		
	037		105		171		024		140		
	016		105		101		042		030		
	025		071		076		101		007		

JNC to IGNOR is not executed. Instead, the number is saved in the L register. If the value of the ASCII character is 100 (40) or greater, the 8080 will ignore the character.

The program that we have just described (Example 7-4) is sometimes called a *software character generator*. Using software, the 8080 has generated characters (dot matrices). These dot matrices were then punched on paper tape. The same type of software can also be used to display characters on a crt or print characters on a printer. If a crt is used, the different dot matrices are used to deflect the electron beam to different points on the face of the crt. If this software is used with a printer that has seven or nine needles or hammers that are used to print characters, then just about any character set can be used with the printer simply by changing the look-up table that the software accesses.

As we mentioned at the beginning of this section, there are also hardware devices (ROMs) that contain look-up tables. In fact, a number of these devices contain look-up tables like the ones that we have described. Some of the ROMs that contain look-up tables (they are called *character generators* by the manufacturers) include the 3257 and 3258 (Fairchild Semiconductor Corporation), the MM4240AA/MM5240AA (National Semiconductor Corporation), the MM6061 and MM6062 (Monolithic Memories, Inc.), and the 2513 and 2516 (Signetics Corporation). For more information about these hardware devices, refer to National Semiconductor's Applications Notes AN-40 and AN-57.

REFERENCES

1. Titus, C. A., Rony, P. R., Larsen, D. G., and Titus, J. A. *8080/8085 Software Design—Book 1.* Howard W. Sams & Co., Inc., Indianapolis, IN, 1978.
2. Leventhal, L. A. "Cut your Processor's Computation Time." *Electronic Design,* August 16, 1977, pp 82-89.
3. Pogge, R. D. "Lookup Tables Provide Quick Logarithmic Calculations." *EDN,* August 5, 1977, pp 87-91.

8

Command Decoders

Quite often, you may want a microcomputer to perform certain operations only under your command. This command might be in the form of setting a set of switches to logic 1s and 0s, or the command might be entered into the microcomputer by means of a teletypewriter or crt keyboard. The microcomputer would have to be programmed so that the key or switch closures are input. When the information is input, the microcomputer can then examine the information to determine the proper actions for the command that was specified. The specified actions can then be performed. This type of software is called a *command decoder.*

Where are command decoders used? There is a command decoder in all BASIC interpreter and compiler programs so that you can use, or enter, commands such as SCR, LIST, LET, FOR, DATA, and IF. There are also command decoders in programs such as *system monitors, debuggers, editors,* and *assemblers.*

SINGLE-LETTER COMMAND DECODERS

The simplest command decoder can cause an action to take place when a single letter or number key on a teletypewriter or crt keyboard is pressed. Each key in the keyboard can represent a specific command and, therefore, a specific sequence of operations to be performed. For a simple system monitor program, the commands listed in Table 8-1 might be useful.

As you can see from Table 8-1, each command is represented by a single letter, and no two commands are represented by the same letter. The command decoder program listed in Example 8-1 will

Table 8-1. A Summary of the Commands for the Command Decoder

Command	Operation
E	Execute a section of the program starting at the symbolic address ENTER.
D	Execute a section of the program starting at the symbolic address DELET.
L	Execute a section of the program starting at the symbolic address LIST.
M	Execute a section of the program starting at the symbolic address MOVE.

actually program the 8080 so that it recognizes (decodes) these four single-letter commands.

In Example 8-1, the 8080 first calls the TTYIN subroutine. In this subroutine, the 8080 will wait for a teletypewriter or crt key to be pressed. When a key is pressed, the 8080 will return from the subroutine with the ASCII character in the A register. Remember, the TTYIN subroutine will also set bit D_7 of the ASCII character to a 0, eliminating any parity bit. The 8080 will also print the character using the TTYOUT subroutine. For a review of tele-

Example 8-1: A Single-Letter Command Decoder for a System Monitor

```
/THIS IS A VERY SIMPLE SINGLE-LETTER COMMAND
/DECODER THAT USES THE CPI INSTRUCTION.

CMDDEC, CALL    /GET A CRT OR TELETYPEWRITER CHARACTER.
        TTYIN
        0
        CPI
        115     /WAS THE CHARACTER AN ASCII M?
        JZ      /YES, THEN EXECUTE THE M = MOVE
        MOVE    /SUBROUTINE.
        0
        CPI
        104     /WAS THE CHARACTER AN ASCII D?
        JZ      /YES, THEN EXECUTE THE D = DELET
        DELET   /SUBROUTINE.
        0
        CPI
        114     /WAS THE CHARACTER AN ASCII L?
        JZ      /YES, THEN EXECUTE THE L = LIST
        LIST    /SUBROUTINE.
        0
        CPI
        105     /WAS THE CHARACTER AN ASCII E?
        JZ      /YES, THEN EXECUTE THE E = ENTER
        ENTER   /SUBROUTINE.
        0
        JMP     /IT WAS NOT AN M,D,L, OR E.
        CMDDEC  /THEREFORE, IGNORE THE CHARACTER AND
        0       /GET ANOTHER CHARACTER.
```

typewriter I/O subroutines, refer to Chapter 3 of *8080/8085 Software Design—Book 1.*[1]

When the 8080 returns from the TTYIN subroutine, it compares the content of the A register to the ASCII value for M (octal 115, hex 4D). If the values are equal, meaning that the M key was pressed, the 8080 executes the JZ instruction to MOVE.

If the D key was pressed, the zero flag is a logic 0 as a result of the CPI 115 instruction. Therefore, the JZ to MOVE is not executed. Instead, the CPI 104 instruction is executed. Since the A register contains the ASCII value for the D key (104), the zero flag will be set to a logic 1 as a result of this comparison. Therefore, the JZ to DELET is executed.

At the symbolic addresses ENTER, DELET, MOVE, and LIST, a sequence of instructions must be stored in memory that perform the desired tasks. After these tasks are performed, the microcomputer should jump to the beginning of the command decoder (CMDDEC) so that another command can be entered and interpreted. Will the 8080 respond to any commands other than E, D, M, and L? No, characters other than E, D, M, or L will be ignored. If the 8080 cannot "match" the key code that was entered with one of the codes used in the command decoder program, the JMP to CMDDEC at the end of the program will be executed.

This type of command decoder is very easy to write and store in the microcomputer, but it does have some limitations. If you need 20 different commands, then there must be 20 compare-immediate instructions and 20 conditional-jump instructions. How many memory locations will be required to store these instructions? The command decoder would require 106 memory locations, because each command requires a two-byte CPI instruction and a three-byte JZ instruction. Six additional memory locations are required for the initial CALL TTYIN instruction and the JMP to CMDDEC instruction at the end of the command decoder program.

A SINGLE-LETTER LIST-BASED COMMAND DECODER

A command decoder program that uses a different decoding technique is listed in Example 8-2. Like many of the forthcoming programs in this chapter, this command decoder uses a list that contains all of the ASCII key values and the addresses to which the 8080 will jump, provided that a valid command is entered. The format of the list is as follows: Each command is represented by a single letter. The ASCII value for the letter is stored in the list and it is immediately followed by the 16-bit (two eight-bit bytes) address for the start of the program that is to be executed when the ap-

Example 8-2: A Flexible Single-Letter Command Decoder

```
/THIS IS  A MORE SOPHISTICATED SINGLE-LETTER
/COMMAND DECODER. IT HAS GREATER FLEXIBILITY.

CMDDEC, LXIH    /REGISTER PAIR H POINTS TO THE
        CMDTAB  /COMMAND/COMMAND-ADDRESS LIST
        0       /OF VALID COMMANDS.
IGNOR,  CALL    /GET A CRT OR TELETYPEWRITER CHARACTER.
        TTYIN
        0
        MOVBA   /SAVE THE ASCII CHARACTER IN B.
CHECK,  MOVAM   /GET A CHARACTER FROM THE LIST.
        CMPB    /COMPARE IT TO THE KEYBOARD CHARACTER.
        JZ
        GETADD  /IF THERE IS A MATCH, FETCH THE ADDRESS THAT
        0       /THE 8080 SHOULD JUMP TO.
        INXH    /INCREMENT THE ADDRESS PAST THE CHARACTER.
        INXH    /THEN INCREMENT IT PAST THE TWO ADDRESS
        INXH    /BYTES STORED AFTER THE CHARACTER.
        MOVAM   /GET A VALUE FROM THE LIST.
        CPI     /IS IT A ZERO (THE LIST
        000     /TERMINATION CHARACTER)?
        JNZ     /A 000 (00) WAS NOT FOUND IN MEMORY,
        CHECK   /SO TRY TO MATCH THE TTY CHARACTER
        0       /WITH ANOTHER LIST CHARACTER.
        JMP     /A 000 (00) WAS FOUND, WHICH MEANS
        IGNOR   /THAT THE ENTIRE LIST WAS SEARCHED.
        0       /THEREFORE, IGNORE THE TTY CHARACTER.
GETADD, INXH    /INCREMENT THE ADDRESS PAST THE CHARACTER.
        MOVEM   /MOVE THE LO BYTE OF THE ADDRESS INTO E.
        INXH    /INCREMENT THE ADDRESS AGAIN.
        MOVDM   /MOVE THE HI BYTE OF THE ADDRESS INTO D.
        XCHG    /THE ADDRESS IS NOW IN H AND L.
        PCHL    /JAM THE NUMBER INTO THE PROGRAM COUNTER (PC).

CMDTAB, 101     /COMMAND CHARACTER IS AN ASCII A.
        125     /THE ADDRESS TO JUMP TO IS
        315     /315 125 (CD55).
        102     /COMMAND CHARACTER IS AN ASCII B.
        232     /THE ADDRESS TO JUMP TO IS
        314     /314 232 (CC9A).
        103     /COMMAND CHARACTER IS AN ASCII C.
        000     /THE ADDRESS TO JUMP TO IS
        370     /370 000 (F800).
        104     /COMMAND CHARACTER IS AN ASCII D.
        000     /THE ADDRESS TO JUMP TO IS
        376     /376 000 (FE00).
        000     /THERE ARE SIX ADDITIONAL MEMORY
        000     /LOCATIONS WHICH CONTAIN ZEROES.
        000     /THESE MEMORY LOCATIONS CAN BE USED
        000     /TO STORE TWO ADDITIONAL COMMANDS
        000     /AND THE APPROPRIATE ADDRESSES.
        000
        000     /THIS IS THE LIST TERMINATOR.
```

propriate command is entered. The LSBY of the address is stored in the list right after the ASCII value, and the MSBY of the address follows the LSBY. Once a list has been generated that contains all of the valid commands and the appropriate 16-bit addresses, a list terminator of 0 must be stored after the last 16-bit address. In this particular example, an additional six memory locations have 0s stored in them so that two additional commands can be added to the list, if required.

At the beginning of Example 8-2, register pair H is loaded with the starting address of the list that contains the valid commands and the 16-bit addresses. The TTYIN subroutine is then called. When a key on the keyboard is pressed, the 8080 inputs the ASCII value for the key, sets bit D_7 to 0, and prints the character. The 8080 returns from the TTYIN subroutine with the seven-bit ASCII value in the A register. This value is then saved in the B register by the MOVBA instruction. At CHECK, the content of the memory location addressed by register pair H is moved to the A register and compared to the content of the B register. These two instructions cause the ASCII value that was input to be compared to the first ASCII value stored in the list.

What happens if the B key on the teletypewriter or crt is pressed? The B key produces the seven-bit ASCII value of 102 (42). When compared to the ASCII value in memory (101, 41), the zero flag is cleared to 0 because the two values are not equal. Therefore, the JZ to GETADD is not executed. Instead, the memory address contained in register pair H is incremented three times. The first time register pair H is incremented it addresses the 125 (55) stored in the list. The second time it addresses the 315 (CD), and the third time it addresses the 102 (42) in the list. The 102 (42) is the second ASCII value stored in the list.

The memory address in register pair H now addresses the next command value that is stored in the list. Since the list may only contain a few commands and their corresponding addresses, the second and all additional commands are compared to the list terminator, 0, used to indicate to the 8080 when it has examined the entire list. Since the next command character has already been fetched from memory (MOVAM), it can easily be compared to the ASCII value that was input, which is stored in the B register.

As a result of the CMPB instruction (register A = 102, register B = 102), and the flag-testing JZ GETADD instruction, the 8080 jumps to the symbolic address GETADD. Since the address in register pair H addresses the matching command stored in the list, it is relatively easy to increment the 16-bit address in register pair H so that the LSBY of the address can be read from memory into the E register. By incrementing the address in register pair H a second time, the MSBY of the address can be read from memory into the D register. The 8080 has now loaded register pair D with the address that it must jump to, since the B command was entered. The XCHG instruction moves this address into register pair H, and the PCHL instruction loads the program counter with this address. The program counter now points to the next instruction to be executed. This instruction is part of the task that the 8080 must perform when the B command is entered.

Suppose that the E key on the keyboard is pressed rather than the B key. What will happen? When the 8080 executes the CMPB instruction at CHECK, it compares the ASCII value for E (105, 45) to the ASCII value for A. These two values are not equal, so the JZ to GETADD is not executed. Instead, the address in register pair H is incremented by 3, so that it addresses the memory location that contains the ASCII value for B (102, 42). This value is then moved to the A register and is compared to 0. Since the value is not equal to 0, indicating that the end of the list has not been reached, the JNZ to CHECK is executed.

At CHECK, the ASCII values for E and B are compared. They are not equal, so the memory address in register pair H is incremented by 3. Register pair H now addresses the memory location that contains the ASCII value for C (103, 43). These two ASCII values are not equal; the memory address in register pair H is incremented again so that it addresses the ASCII value for D stored in the list, and so on until either the correct ASCII value or the list terminator is reached. In this case, there is no ASCII value for E stored in the list, so the list terminator is found. This value is moved from memory to register A and is compared to 0 by the CPI 000 instruction. Because of this, the JNZ to CHECK is not executed. Instead, the 8080 jumps to IGNOR. This causes the E key (command) to be ignored. The 8080 then calls the TTYIN subroutine so that another command can be entered. Why was the E command ignored? The E command was ignored because there was no ASCII value for E stored in the list, which means that the E command is *undefined;* i.e., no operations were assigned to the E command.

What other commands are ignored? All other commands, except A, B, C, and D are ignored. How many memory locations are required to add a new command to the list? Only three memory locations are required. One memory location is used to store the ASCII value for the command, and the other two memory locations contain the 16-bit address that the 8080 must jump to when the command is entered. In Example 8-2 seven memory locations contain 0s, so two additional commands can be added to the list. Note that we still must have a list terminator stored in memory after the last 16-bit address. For this reason, seven memory locations contain 0s in Example 8-2.

At the end of the command decoder program the XCHG and PCHL instructions cause the 8080 to "jump" to the specified memory location. What other two single-byte instructions would cause the same action to take place? The instructions PUSHD and RET would also cause the 8080 to "jump" to the proper address. These instructions cause the 16-bit address in register pair D to be stored on the stack. The RET instruction pops this address off of the stack and into the program counter.

How many memory locations are required for the complete command decoder program if just two commands have to be decoded? This is the amount of memory required to store the command decoder program *and* the list. The basic command decoder program requires 30 memory locations, regardless of how many commands can be decoded. For two commands, an additional six memory locations are required to store the two ASCII values and the two 16-bit addresses. In addition, a list termination character (0) must also be stored in memory. Therefore, a total of 37 memory locations are required by this type of command decoder to decode just two commands. Only 16 memory locations are required to decode two commands if the CPI-type command decoder (Example 8-1) is used. Is there a point where the list-type command decoder will require less memory than the CPI-type command decoder? Yes. This point can be calculated using the formula shown in Table 8-2.

Table 8-2. Calculating the Storage Requirements for the Two Different Types of Command Decoders

CPI Method	List Method
$S = 6 + (C \times 5)$	$S = 31 + (C \times 3)$
C = number of commands. S = number of memory locations required.	

From Table 8-2, we can determine that if 13 or more commands are required, the list method becomes more memory-efficient. Note that it is relatively easy to add new commands to both types of command decoders. In the CPI-based command decoder, you would simply place 0s in memory after the JMP to CMDDEC instruction. To add new commands to the command decoder, you would write over the JMP to CMDDEC instruction with the appropriate CPI and JZ instructions. After the last sequence of these two instructions, you would have to store a JMP CMDDEC instruction. For the list-based command decoder, you would simply have to save some 0s after the last ASCII value and its associated 16-bit address. These memory locations would then be used to store the ASCII values for new commands and 16-bit addresses. Of course, a 0 would have to be stored in memory after the last 16-bit address as the list terminator.

A SINGLE-LETTER COMMAND DECODER THAT USES TWO LISTS

If you want, a command decoder can also be written that accesses two lists. One list would contain the ASCII values for the valid commands, and the other list would contain the addresses that the 8080 jumps to when one of the valid commands is entered. The program listed in Example 8-3 uses these two types of lists.

In Example 8-3, the 8080 loads register pair D with the address of the list that contains the ASCII values for the valid commands, and register pair H is loaded with the address of the list that contains the 16-bit addresses for the valid commands. In many respects, this command decoder is the same as the one listed in Example 8-2. The only differences are that one program uses one list and the other program uses two lists.

As you can see, the program that uses two lists is a little longer than the command decoder that uses one list. One reason is that two LXI-type instructions have to be executed at the beginning of the command decoder listed in Example 8-3. However, at GETADD (Example 8-3), the address in register pair H does not have to be incremented before the E register can be loaded with the LSBY of the proper 16-bit address. This means that Example 8-3 requires only two more memory locations for storage than Example 8-2. Note that in Example 8-3, only one of the two lists (CMDTAB) requires a list terminator. This is because if the 8080 gets to the end of this list, it had better be at the end of the other list. From these two examples, you should determine that there is really no advantage in using a command decoder that uses two lists.

MULTIPLE-CHARACTER COMMAND DECODERS

Suppose that you need to write a single-letter command decoder for the commands EXIT, ENTER, EXTRACT, and EQUALIZE. As you would expect, we cannot assign the E key to all four of these commands. Some possible key/command assignments are listed in Table 8-3.

It would be very easy to write a single-letter command decoder that uses the key/command assignments listed in Table 8-3. However, it may be difficult to remember that the Q key represents the EQUALIZE command and that the N key represents the ENTER

Example 8-3: A Two-List Single-Letter Command Decoder

```
/THIS COMMAND DECODER USES TWO SEPARATE LISTS.
/ONE LIST CONTAINS THE VALID COMMANDS AND
/THE OTHER LIST CONTAINS THE COMMAND ADDRESSES.

CMDDEC, LXID     /REGISTER PAIR D POINTS TO THE
        CMDTAB   /LIST OF VALID COMMANDS.
        0
        LXIH     /REGISTER PAIR H POINTS TO THE
        CMDADD   /LIST ADDRESS.
        0
IGNOR,  CALL     /GET A CRT OR TELETYPEWRITER CHARACTER.
        TTYIN
        0
        MOVBA    /SAVE THE KEYBOARD CHARACTER IN B.
CHECK,  LDAXD    /GET ONE OF THE ASCII CHARACTERS FROM THE LIST.
        CMPB     /COMPARE IT TO THE KEYBOARD CHARACTER.
        JZ
        GETADD   /IF THERE IS A MATCH, FETCH THE ADDRESS THAT
        0        /THE 8080 SHOULD JUMP TO.
        INXD     /INCREMENT TO THE NEXT CHARACTER.
        INXH     /INCREMENT TO THE NEXT ADDRESS
        INXH     /IN THE "CMDADD" LIST.
        LDAXD    /GET A VALUE FROM THE COMMAND LIST.
        CPI      /IS IT A ZERO (THE LIST
        000      /TERMINATION CHARACTER)?
        JNZ      /A 000 (00) WAS NOT FOUND IN MEMORY,
        CHECK    /SO TRY TO MATCH THE TTY CHARACTER
        0        /WITH ANOTHER LIST CHARACTER.
        JMP      /A 000 (00) WAS FOUND, WHICH MEANS
        IGNOR    /THAT THE ENTIRE LIST HAS BEEN SEARCHED.
        0        /THEREFORE, IGNORE THE TTY CHARACTER.
GETADD, MOVEM    /MOVE THE LO BYTE OF THE ADDRESS INTO E.
        INXH     /INCREMENT THE ADDRESS AGAIN.
        MOVDM    /MOVE THE HI BYTE OF THE ADDRESS INTO D.
        XCHG     /THE ADDRESS IS NOW IN H AND L.
        PCHL     /JAM THE NUMBER INTO THE PROGRAM COUNTER (PC).

CMDTAB, 101      /COMMAND CHARACTER IS AN A.
        102      /COMMAND CHARACTER IS A B.
        103      /COMMAND CHARACTER IS A C.
        104      /COMMAND CHARACTER IS A D.
        000      /ASCII CODE FOR AN ADDITIONAL COMMAND.
        000      /ASCII CODE FOR AN ADDITIONAL COMMAND.
        000      /TERMINATOR.

CMDADD, 125      /ADDRESS FOR THE A COMMAND.
        315
        232      /ADDRESS FOR THE B COMMAND.
        314
        000      /ADDRESS FOR THE C COMMAND.
        370
        000      /ADDRESS FOR THE D COMMAND.
        376
        000      /FOR AN ADDITIONAL ADDRESS (COMMAND).
        000
        000      /FOR AN ADDITIONAL ADDRESS (COMMAND).
        000
```

Table 8-3. Key Assignments for the EXIT, EXTRACT, ENTER, and EQUALIZE Commands

Key	Command
E	EXIT
N	ENTER
X	EXTRACT
Q	EQUALIZE

command. In fact, the E and X keys can be easily confused, because both the EXIT and EXTRACT commands start with EX. To eliminate this confusion, we can write a command decoder that uses four-character commands. Thus, to exit from a program, the EXIT command would be entered. To execute the EXTRACT section of a program, the command EXTR would be entered. A summary of the possible four-character commands for these four operations is listed in Table 8-4. The command decoder that uses these four-character commands is listed in Example 8-4.

Table 8-4. A Summary of the Four-Character Commands

Command	Operation
EXIT	EXIT
EXTR	EXTRACT
ENTR	ENTER
EQUA	EQUALIZE

As you can see immediately, the command decoder listed in Example 8-4 is far more complex than any of our previous command decoder examples. The four-character command now has to be stored in R/W memory rather than in a series of registers. Once the command has been entered, the 8080 has to search for a node in the list that matches all four of the characters that were entered on the teletypewriter keyboard, in the same order that they were entered. However, one advantage of this type of command decoder is that commands entered are very similar to the operations that the microcomputer actually performs.

In the first part of Example 8-4, the 8080 loads register pair H with a R/W memory address. This address has been assigned the symbolic address CMD. The C register is then loaded with the number of characters that can be entered on the keyboard and stored in R/W memory, starting at CMD. At CMDIN, the TTYIN subroutine is called, so that the 8080 can input an ASCII value. When the 8080 returns from TTYIN, the character will have been printed on the printer, and bit D_7 of the ASCII value will be 0. The content of the A register is then compared to the seven-bit value for a CTRL/C code. If a CTRL/C code is entered, the 8080 jumps to CMDDEC. This means that the CTRL/C code is used to abort the command that was in the process of being entered. By generating the CTRL/C code, we can begin to enter a new and, we hope, correct command. If the value entered was not a CTRL/C, the ASCII value in the A register is saved in the memory location addressed by register pair H. The address is then incremented and the count in register C is decremented. This means that the JNZ to CMDIN is executed three times, so that four characters can be entered and stored in R/W memory. The fourth time a character is entered, the content of the C register is decremented to 0, so the JNZ to CMDIN is not executed.

Register pair H is loaded with the list address at DECODE, and the C register is loaded with the number of characters in each command (four). The 8080 then moves a value from the list addressed by register pair H to the A register, and compares the value to 0. What does this sequence of instructions do? It detects the end of the list. If a 0 is read from the list, then a match has not occurred between the command that was typed into the microcomputer and any of the nodes in the list. If the 8080 has not reached the end of the list, register pair D is loaded with the starting address of the R/W memory locations in which the command that was entered from the keyboard is stored.

At NXTCHR, the first character input from the teletypewriter is loaded into the A register. This value is then compared to the first value stored in the list. If the two values are not equal, the 8080 jumps to NXTCMD, where it must find the beginning of the next command in the list. If the first two ASCII values are the same (they match), the 8080 will attempt to match the next three consecutive characters, one at a time. The 8080 increments the two memory addresses by executing the INXH and INXD instructions, and the character count in the C register is decremented by 1. If the content of the C register is nonzero, the JNZ to NXTCHR is executed. This means that four matches have not yet occurred. If the content of the C register is decremented to 0, then four consecutive matches have occurred between the ASCII values of the command entered and the ASCII values of one of the commands in the list.

When a complete four-character match takes place, the 8080 must fetch the proper 16-bit address from the list and transfer program execution to that address. This 16-bit address is stored in the list, LSBY first, immediately after the four ASCII values (characters) for the command. So after a four-character match takes place, the contents of the next two consecutive memory locations are loaded into register pair D. The XCHG instruction loads register pair H

Example 8-4: A Four-Letter-per-Command Command Decoder

```
/THIS IS A FOUR-LETTER-PER-COMMAND
/COMMAND DECODER.

CMDDEC, LXIH    /REGISTER PAIR H IS USED AS THE
        CMD     /MEMORY ADDRESS TO STORE THE
        0       /COMMAND TYPED IN.
        MVIC    /REGISTER C IS USED TO COUNT
        004     /THE NUMBER OF CHARACTERS.
CMDIN,  CALL    /GET A TELETYPEWRITER CHARACTER.
        TTYIN   /ECHO IT AND RETURN WITH IT
        0       /IN THE A REGISTER.
        CPI     /WAS A CTRL/C GENERATED BY THE
        003     /KEYBOARD? IF SO, ABORT.
        JZ      /REINITIALIZE THE MEMORY ADDRESS IN
        CMDDEC  /REGISTER PAIR H AND THE CHARACTER
        0       /COUNT IN THE C REGISTER.
        MOVMA   /SAVE THE COMMAND CHARACTER IN MEMORY.
        INXH    /INCREMENT THE MEMORY ADDRESS.
        DCRC    /DECREMENT THE CHARACTER COUNTER.
        JNZ     /4 CHARACTERS YET? NO, GET ANOTHER.
        CMDIN
        0
DECODE, LXIH    /THE COMMAND HAS BEEN TYPED IN, NOW
        CMDTAB  /SEE WHAT IT IS. REGISTER PAIR H
        0       /ADDRESSES THE COMMAND LIST.
AGAIN,  MVIC    /REGISTER C IS THE CHARACTER-
        004     /PER-COMMAND COUNTER.
        MOVAM   /GET A LIST CHARACTER.
        CPI     /IS IT THE LIST TERMINATOR?
        000
        JZ      /YES, THE ENTIRE LIST HAS BEEN
        CMDDEC  /EXAMINED AND NO MATCHES OCCURRED,
        0       /SO IGNORE THIS COMMAND AND GET ANOTHER.
        LXID    /REGISTER PAIR D ADDRESSES MEMORY
        CMD     /WHERE THE COMMAND TYPED IN
        0       /IS STORED.
NXTCHR, LDAXD   /GET A COMMAND CHARACTER.
        CMPM    /COMPARE IT WITH A LIST CHARACTER.
        JNZ     /NO MATCH, FIND THE NEXT COMMAND
        NXTCMD  /IN THE LIST.
        0
        INXH    /THE LETTERS MATCHED, TRY THE
        INXD    /NEXT TWO. INCREMENT BOTH ADDRESSES.
        DCRC    /DECREMENT THE CHARACTER COUNTER.
        JNZ     /ALL FOUR COMPARED YET? NO,
        NXTCHR  /COMPARE THE NEXT CHARACTERS.
        0
        MOVEM   /GET THE ADDRESS INTO
        INXH    /REGISTER PAIR D.
        MOVDM
        XCHG    /PUT THE ADDRESS INTO H AND L.
        PCHL    /JUMP TO THAT ADDRESS.
NXTCMD, INRC    /INCREMENT THE CHARACTER COUNT
        INRC    /BY 2 BECAUSE OF THE ADDRESS BYTES.
```

```
NOTYET, INXH    /INCREMENT THE LIST ADDRESS.
        DCRC    /GET TO THE NEXT COMMAND YET? NO,
        JNZ     /KEEP INCREMENTING THE ADDRESS.
        NOTYET
        0
        JMP     /H AND L POINT TO THE NEXT COMMAND
        AGAIN   /IN THE LIST, SO TRY TO MATCH
        0       /THE COMMAND WITH THE LIST ENTRY.

CMDTAB, 105     /ASCII E IN "EXIT"
        130     /X
        111     /I
        124     /T
        125     /LO ADDRESS
        315     /HI ADDRESS
        105     /ASCII E IN "EXTR"
        130     /X
        124     /T
        122     /R
        232     /LO ADDRESS
        314     /HI ADDRESS
        105     /ASCII E IN "ENTR"
        116     /N
        124     /T
        122     /R
        000     /LO ADDRESS
        370     /HI ADDRESS
        105     /ASCII E IN "EQUA"
        121     /Q
        125     /U
        101     /A
        000     /LO ADDRESS
        376     /HI ADDRESS
        000     /LIST TERMINATOR
```

with this address, and the PCHL instruction loads the program counter with the content of register pair H, the address pointing to the section of the program to which the command has directed the program flow.

Why is there no INXH instruction just before the MOVEM instruction after NXTCHR? The last time through the loop, the addresses in both register pairs D and H are incremented by 1. The content of the C register is then decremented to 0. At this point in the execution of the program, register pair H addresses the memory location that contains the appropriate LSBY of the 16-bit address that program execution is going to be transferred to.

If the 8080 cannot match the command that was entered on the keyboard with the first four-character command in the list, the JNZ to NXTCMD is executed. At NXTCMD, the content of the C register is incremented by 2. Why is this done? Remember, the count in the C register only reflects the number of *characters* in the command. However, the list contains not only *four* characters per command, but also *two* eight-bit address bytes. These address bytes have to be skipped by the 8080 so that it can get to the next command in the list. Therefore, the content of register C is incremented by 2, so that the 8080 takes into account the two memory locations used to store this 16-bit address. After the two INRC instructions are executed, the memory address in register pair H is incremented, while the count in the C register is decremented, so that register pair H will address the memory location that contains the first character in the next command in the list.

What will happen if the command EXTR is entered? The microcomputer matches the E and the X of the EXTR command to the first node in the list, EXIT. However, the ASCII values for T and I are not equal, so the 8080 executes the JNZ to NXTCMD. At this point, the C register contains 002 (02). At NXTCMD, this number is incremented to 004 (04), and register pair H is incremented so that it addresses the memory location that contains the T in the EXIT command. The content of the C register is then decremented from 004 to 003 (04 to 03). To skip over the 16-bit address stored in the list after the EXIT command, register pair H is incremented to the LSBY of the address, while the C register is decremented to 002 (02). To skip over the MSBY of the address, register pair H is incremented, and the C register is decremented to 001 (01). Finally, the address in register pair H is incremented so that it addresses the memory location that contains the first character in the next command. At this point, the content of the C register is decremented to 0. The 8080 then jumps to AGAIN.

At AGAIN, register C is loaded with the character count 004 (04), and the 8080 checks to see if the end of the list has been reached. The value 105 (45) is moved from the list to the A register and, since this value is not equal to 0, the 8080 continues with the search. Therefore, just before NXTCHR, register pair D is loaded with the address for the first character that was entered on the keyboard and saved in R/W memory. The 8080 then compares the command entered, EXTR, to the command addressed by register pair H, EXTR. As expected, a four-character match occurs. The address for EXTR (314 232, CC9A) is then read from the list into register pair D. This address is exchanged with the content of register pair H, and is then loaded into the program counter.

VARIABLE-LENGTH COMMAND DECODERS

The last major improvement that can be made to the four-character-per-command command decoder (Example 8-4) is to make all of the commands variable in length. That is, the commands can contain as many or as few characters as required. For the commands, EXIT, EXTRACT, ENTER, and EQUALIZE, all of the letters in the commands would be "used" by the command decoder. Table 8-5 compares the variable-length and fixed-length commands.

Table 8-5. A Comparison of Variable-Length and Fixed-Length Commands

Variable Length	Fixed Length
EXIT	EXIT
EXTRACT	EXTR
ENTER	ENTR
EQUALIZE	EQUA

If variable-length commands are used, the *structure* of the list used by the command decoder software will have to change. This means that instead of storing just the first four letters of the command in the list, all of the letters in the command will be stored in the list. As you can see, all of the commands contain a different number of letters. If the structure of the list is changed, then the command decoder program will also have to be changed. The command-input section of the program has to be changed so that any number of characters can be entered and stored in R/W memory. Only when a special termination character is entered should the actual "matching" section of the program be executed. Of course, there is a practical limit to the number of characters in a command. It is doubtful that any command longer than 20 characters would ever be used, just because it is difficult to remember commands that are this long. To terminate the string of characters that are entered into the microcomputer, the RETURN key (ASCII 015, 0D) will be used. It is an arbitrary choice. Therefore, only after the RETURN key is pressed will the 8080 attempt to match the characters that were entered with a string of characters in the list.

Since the length of the command is variable, can the counter technique be used to determine when the end of the command in the list has been reached? It can be if we store *the number of characters in each command* before each command in the list. For instance, the number 004 (04) is stored in front of the ASCII characters for the EXIT command. For EXTRACT, the number 007 (07) is stored in front of the ASCII characters; for ENTER, the number 005 (05) is used; and for EQUALIZE, the number 010 (08) is used. Of course, this method requires that you know how many characters will be used in each command, since this number would have to be stored in the list just before the first character of the command. A portion of this new list is shown in Example 8-5.

Example 8-5: The Variable-Length Command List Structure

```
CMDTAB, 004       /NUMBER OF LETTERS IN "EXIT"
        105       /ASCII E
        130       /X
        111       /I
        124       /T
        125       /LO ADDRESS
        315       /HI ADDRESS
        007       /NUMBER OF LETTERS IN "EXTRACT"
        105       /ASCII E
        130       /X
        124       /T
        122       /R
        101       /A
        103       /C
        124       /T
        232       /LO ADDRESS
        314       /HI ADDRESS
         •
         •
```

Before the actual comparison of the command entered and one of the commands in the list is performed, the character count must be moved from the list into the C register. In all other respects, the instructions that affect the C register remain the same. However, there is one additional problem associated with a variable-length-command command decoder. Suppose that there are only two commands in the list. They are stored using our new format:

```
CMDTAB, 002 /NUMBER OF LETTERS IN "GO"
        107 /ASCII G
        117 /O
        107 /LO ADDRESS
        232 /HI ADDRESS
        004 /NUMBER OF LETTERS IN "GOOD"
        107 /ASCII G
        117 /O
        117 /O
        104 /D
        345 /LO ADDRESS
        246 /HI ADDRESS
        000 /LIST TERMINATOR
```

For each command, GO and GOOD, the number of characters in the command is stored in front of the first ASCII character in the command (the "test string"). These are 002 and 004, respectively. The ASCII values for the characters are then stored in the list, followed by the appropriate 16-bit address. At the end of the list, the list termination character is stored.

Now suppose that the GOOD command is entered. Based on the preceding list, to which address will the 8080 jump: 232 107 (9A47) or 246 345 (A6E5)? The microcomputer will "incorrectly" jump to address 232 107 (9A47). The reason for this is that the microcomputer would successfully match the two-character GO command in the list to the first two characters in the GOOD command that was entered on the keyboard. Since there are only two characters in the GO command, the count contained in the C register would be decremented from 2 to 0. Since the match was "successful," the address 232 107 (9A47) would be fetched from memory and used as the starting address for the GOOD command that was entered. Is there a simple solution to this problem? Yes, there is. All the microcomputer has to do is to determine whether the end of the command entered (which is stored in R/W memory) has been reached when the end of the ASCII string in the list has been reached. In the previous example, the microcomputer would match the GO command in the list to the GO in the GOOD command that was entered, but when the microcomputer checked to see if the end of the GOOD command was reached, it would find that it still had the OD of the GOOD command yet to be matched. In other words, the microcomputer would not find the termination character, RETURN, when it examined the memory locations where the GOOD command was stored when it was entered. (The microcomputer would find the RETURN if the GO command, alone, had been entered.)

The variable-length command decoder program is listed in Example 8-5. Note that in the CMDIN section of the program, the ASCII value for the RETURN key *is* stored in memory, after the command.

As you have seen, command decoders make it very easy for us to direct the sequence of operations performed by the microcomputer. In small systems, we might have only four or five commands, where each command is represented by a single letter or number. If the number of operations that the microcomputer must perform increases, the complexity of the commands (the number of alphanumeric characters in each command) may increase. If the commands increase in complexity, then a multiple-character command decoder will probably have to be used. As we have seen, this type of command decoder would be able to decode commands such as CLOSE VALVE 3 or PUMP 1 ON.

All of the command decoders that we have discussed in this chapter can be thought of as *interpreters*. We enter a string of alphanumeric characters into the microcomputer, and the microcomputer *interprets* the string of characters and then performs operations *specific* for that command. Once these operations have been performed, the microcomputer returns to the command decoder program so that another command can be entered and interpreted.

Example 8-6: A Variable-Length-Command Command Decoder

```
/THIS  IS THE VARIABLE-LENGTH-
/COMMAND, COMMAND DECODER

CMDDEC, LXIH    /REGISTER PAIR H IS USED AS THE
        CMD     /MEMORY ADDRESS TO STORE THE
        0       /COMMAND TYPED IN.
CMDIN,  CALL    /GET A TELETYPEWRITER CHARACTER.
        TTYIN   /ECHO IT AND RETURN WITH IT
        0       /IN THE A REGISTER.
        MOVMA   /SAVE THE COMMAND CHARACTER IN MEMORY.
        INXH    /INCREMENT THE MEMORY ADDRESS.
        CPI     /WAS THE CHARACTER SAVED IN MEMORY
        015     /THE TERMINATOR (A CARRIAGE RETURN)?
        JNZ     /NO, IT WAS NOT A CARRIAGE RETURN,
        CMDIN   /SO GET ANOTHER COMMAND CHARACTER.
        0
DECODE, LXIH    /THE COMMAND HAS BEEN TYPED IN, NOW
        CMDTAB  /SEE WHAT IT IS. REGISTER PAIR H POINTS
        0       /TO THE COMMAND/COMMAND-ADDRESS LIST.
AGAIN,  XRAA    /SET THE A REGISTER TO ZERO.
        CMPM    /COMPARE THE CONTENT OF MEMORY TO REGISTER A.
        JZ      /THE CONTENT OF MEMORY IS ZERO, SO THE
        CMDDEC  /ENTIRE LIST HAS BEEN EXAMINED AND NO
        0       /MATCHES OCCURRED, SO GET ANOTHER COMMAND.
        MOVCM   /MEMORY NOT ZERO, SO IT'S THE CHARACTER COUNT.
        INXH    /INCREMENT TO THE FIRST LETTER IN THE LIST.
        LXID    /REGISTER PAIR D ADDRESSES MEMORY
        CMD     /WHERE THE COMMAND TYPED IN
        0       /IS STORED.
NXTCHR, LDAXD   /GET A COMMAND CHARACTER.
        CMPM    /COMPARE IT WITH A LIST CHARACTER.
        JNZ     /NO MATCH, FIND THE NEXT COMMAND
        NXTCMD  /IN THE LIST.
        0
        INXH    /THE LETTERS MATCHED, TRY THE
        INXD    /NEXT TWO. INCREMENT BOTH ADDRESSES.
        DCRC    /DECREMENT THE CHARACTER COUNTER.
        JNZ     /COMPARED ALL CHARACTERS YET? NO,
        NXTCHR  /COMPARE THE NEXT TWO CHARACTERS.
        0
        LDAXD   /IS THERE A CARRIAGE RETURN AT
        CPI     /THE END OF THE COMMAND JUST
        015     /TYPED IN?
        JNZ     /NO. THEREFORE, THE PROPER COMMAND
        NXTCMD  /HAS NOT BEEN FOUND IN THE
        0       /LIST. TRY THE NEXT COMMAND.
        MOVEM   /GET THE ADDRESS INTO
        INXH    /REGISTER PAIR D.
        MOVDM
        XCHG    /PUT THE ADDRESS INTO H AND L.
        PCHL    /JUMP TO THAT ADDRESS.
NXTCMD, INRC    /INCREMENT THE CHARACTER COUNT
        INRC    /BY 2 BECAUSE OF THE ADDRESS BYTES.
```

```
NOTYET, INXH    /INCREMENT THE LIST ADDRESS.
        DCRC    /GET TO THE NEXT COMMAND YET? NO,
        JNZ     /KEEP INCREMENTING THE ADDRESS.
        NOTYET
        0
        JMP     /H AND L POINT TO THE NEXT COMMAND
        AGAIN   /IN THE LIST, SO TRY TO MATCH
        0       /THE COMMAND WITH THE LIST ENTRY.

CMDTAB, 004     /NUMBER OF LETTERS IN "EXIT"
        105     /ASCII E
        130     /X
        111     /I
        124     /T
        125     /LO ADDRESS
        315     /HI ADDRESS
        007     /NUMBER OF LETTERS IN "EXTRACT"
        105     /ASCII E
        130     /X
        124     /T
        122     /R
        101     /A
        103     /C
        124     /T
        232     /LO ADDRESS
        314     /HI ADDRESS
        005     /NUMBER OF LETTERS IN "ENTER"
        105     /ASCII E
        116     /N
        124     /T
        105     /E
        122     /R
        000     /LO ADDRESS
        370     /HI ADDRESS
        010     /NUMBER OF LETTERS IN "EQUALIZE"
        105     /ASCII E
        121     /Q
        125     /U
        101     /A
        114     /L
        111     /I
        132     /Z
        105     /E
        000     /LO ADDRESS
        376     /HI ADDRESS
        000     /THIS IS THE LIST TERMINATOR
```

REFERENCE

1. Titus, C. A., Rony, P. R., Larsen, D. G., and Titus, J. A. *8080/8085 Software Design—Book 1.* Howard W. Sams & Co., Inc., Indianapolis, IN, 1978.

9

System Monitors

As you know, the 8080 or 8085 microprocessor integrated circuit does not come from the manufacturer with a 50- or 100-instruction program already programmed into it. However, it is "programmed" to perform specific operations when a particular instruction is read from memory into the instruction register (IR). Suppose that you already have an 8080 microcomputer system that has 1024 words of R/W memory. This memory is wired for the addresses 000 000 through 003 377 (0000 through 03FF). The memory is wired for these addresses because when the RESET pin of the 8080 is taken to a logic 1, the 8080 program counter is cleared, and the instruction stored in memory location 000 000 (0000) is read into the IR and executed. Now you want to start programming the microcomputer. Unfortunately, you have no peripherals that can be used to enter a program into the microcomputer.

HARDWIRED FRONT PANELS

Microcomputer designers realized that the lack of an I/O device could be a problem; one of their solutions has been to incorporate a *hardwired front panel* in the microcomputer system. A hardwired front panel enables you to directly access the memory. That is, you can use the front panel to *deposit* information into memory and *examine* information stored in memory. The front panel is hardwired to the address and data buses of the microcomputer. There are no input or output instructions that are executed to access the front panel. This means that the front panel has to generate a 16-bit memory address so that the user can deposit the state of eight switches (logic 1 or logic 0) into a specific memory location. The front panel also has to pulse the $\overline{\text{MEMW}}$ and $\overline{\text{MEMR}}$ lines to write information into memory or to read information from memory.

Since the front panel is using the address and data buses of the microcomputer, can the 8080 be executing a program while you deposit information from the front panel into memory? No. In fact, the front panel has to *lock-out*, or prevent, the 8080 from using these buses while the front panel is in use. On the other hand, while the 8080 is executing a program, the front panel must be prevented from using these buses.

All of these functions require a considerable amount of hardware, and hardwired front panels are expensive to manufacture. The schematic diagram of the hardwired front panel for the E & L Instruments, Inc. MD-1 microcomputer is shown in Figs. 9-1 and 9-2. Hardwired front panels are also used on PDP-8s (Digital Equipment Corporation), the Intellec Series of microcomputers (Intel Corporation), and some of the MITS (PERTEC) and IMS Associates Inc. (IMSAI) microcomputers.

By using a hardwired front panel, you can enter instruction op codes or data values directly into memory. No software is required to do this. Once the instructions (and data, if required) are saved in memory, the 8080 can be reset and it will begin to execute the program that is stored in memory, starting at memory location 000 000 (0000). However, with this type of front panel, it will take a considerable amount of time to enter a 50- or 60-instruction program.

To eliminate the high costs of hardwired front panels and reduce the effort required to enter information into the microcomputer, many manufacturers have microcomputer systems with *software-driven "front panels."* This type of front panel is far more prevalent today and it can be found on the KIM (MOS Technology), the MMD-1 (E & L Instruments), the SDK-85 (Intel Corporation), and the H8 (Heath Company). The most prevalent type of software-driven front panel consists of a 16- to 25-key keyboard and some seven-segment LED displays. These devices are interfaced to the microcomputer as either accumulator I/O or memory-mapped I/O devices. The microcomputer also *has to have* some nonvolatile read-only memory (ROM) so that the program that "drives" the keyboard and LED displays is always present in the microcomputer system.

This means that when the power to the microcomputer is turned off, the system monitor program stored in ROM is not lost; it remains in the ROM. If the system monitor is stored in R/W memory, the program will be lost if the power is turned off. This type of memory requires electrical power to keep the information stored; ROM does not. Both the Commodore PET and the Radio Shack TRS-80 use ROM to store a BASIC interpreter. By their doing this, you do not have to load an audio cassette into the microcomputer to get it "up and running."

The system monitor forms the "heart" or "kernel" of the microcomputer system. Without the system monitor program, you would not be able to enter instruction op codes or data values into the microcomputer on the keyboard and have the values displayed on LEDs or stored in R/W memory.

GENERAL SYSTEM MONITOR CHARACTERISTICS

The operations that a simple system monitor can perform are summarized in Table 9-1. If a system monitor, combined with the software-driven front panel, can perform all of these operations, then it can perform all of the operations that a hardwired front panel can perform, but under software control. One of the simplest, but not necessarily the least expensive, software-driven front panels would consist of an eight-bit ASCII keyboard and six to nine seven-segment LED displays. We need nine displays if address and data information is displayed in octal, and six displays if the same information is displayed in hexadecimal. If the octal (hexadecimal) numbering system is used, six (four) displays will be used to display a 16-bit memory address, and three (two) will be used to display the content of memory at that address. We will use an ASCII keyboard initially because it is a common peripheral device. We will assume that the interface logic of the keyboard *encodes* and *debounces* the key closures.

Table 9-1. A Command Summary for a Simple System Monitor

1. Specify any 16-bit memory address.
2. Examine the content of any memory location.
3. Change the content of any memory location.
4. Begin executing a program at any memory address.
5. Increment the current memory address.

What operations do we want the system monitor to perform? We want to be able to specify either a 16-bit address or an eight-bit data word. The data word will represent either an instruction op code or an actual data value. To specify that the information to be entered is a 16-bit address, we will have to press the A key, followed by six octal or four hexadecimal digits. Once the entire 16-bit address has been entered, it is displayed on the seven-segment "address" displays. The content of the memory location, whose address is displayed, must then be displayed on the seven-segment "data" displays. At this point, we may be satisfied with the data in this mem-

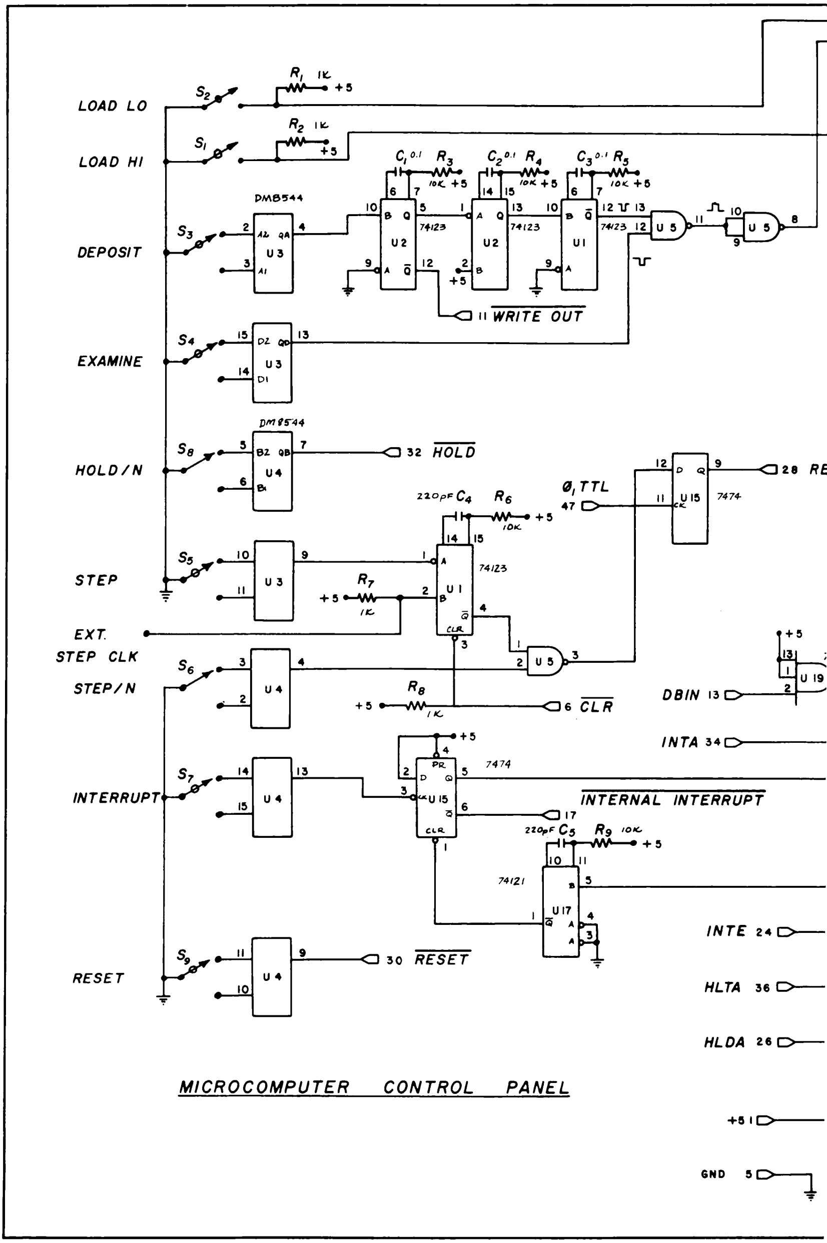
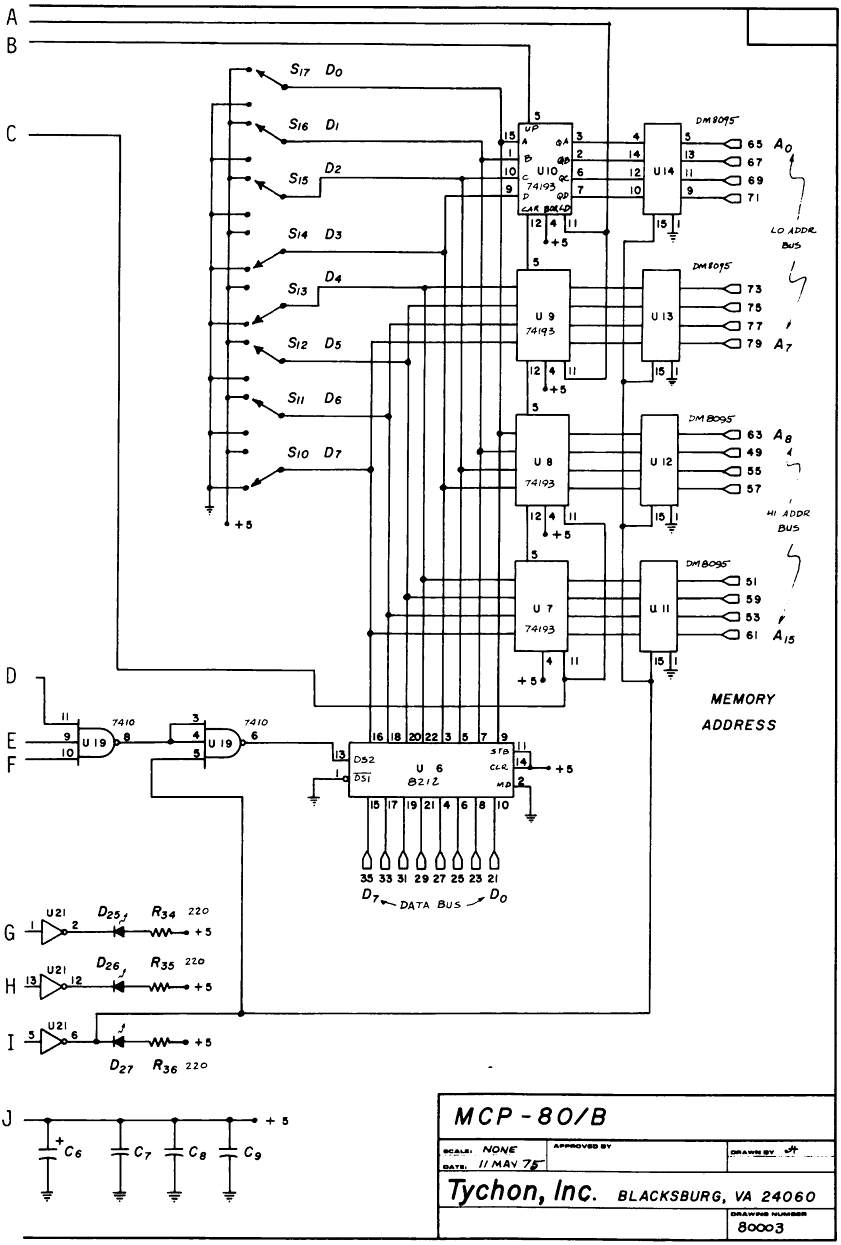
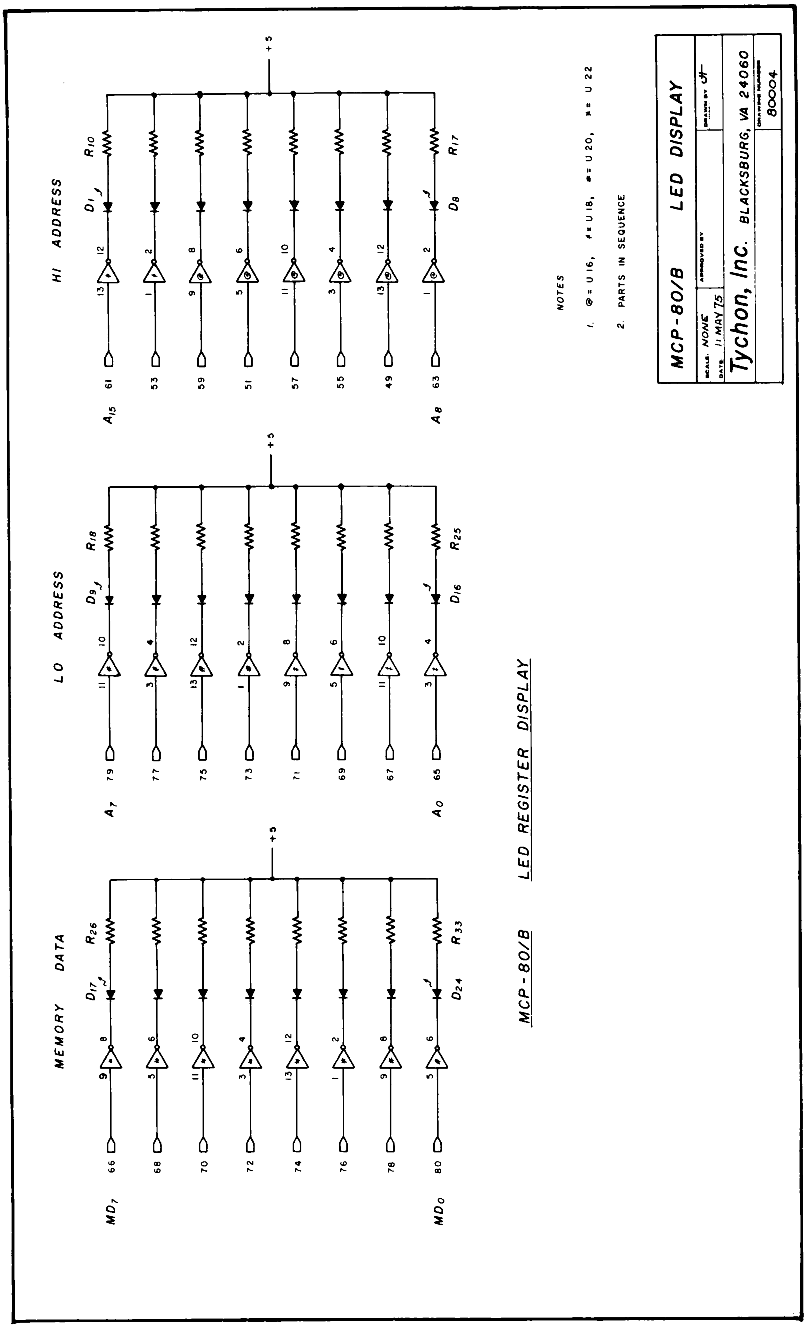

LOAD LO
LOAD HI
DEPOSIT
EXAMINE
HOLD/N
STEP
EXT. STEP CLK
STEP/N
INTERRUPT
RESET
S2
S1
S3
S4
S8
S5
S6
S7
S9
R1 1K
R2 1K
+5
DM8544
U3
U4
DM8544
C1 0.1
R3 10K
C2 0.1
R4 10K
C3 0.1
R5 10K
U2
74123
U1
U5
11 WRITE OUT
32 HOLD
Ø1 TTL
47
U15
7474
28 RE
220pF C4
R6 10K
R7 1K
74123
6 CLR
R8 1K
DBIN 13
INTA 34
U19
INTERNAL INTERRUPT
17
220pF C5
R9 10K
74121
U17
INTE 24
HLTA 36
HLDA 26
30 RESET
+5 1
GND 5
MICROCOMPUTER CONTROL PANEL
DIETZGEN MASTER FORM 100MP
PRINTED ON DIETZGEN 100M AGEPROOF VELLUM

Fig. 9-1. Schematic diagram of the hardwired front panel for the


A
B
C
D
E
F
G
H
I
J
S17 D0
S16 D1
S15 D2
S14 D3
S13 D4
S12 D5
S11 D6
S10 D7
+5
U10 74193
UP
A
B
C
D
QA
QB
QC
QD
CAR BOR LD
DM8095
U14
65 A0
67
69
71
LO ADDR BUS
U9 74193
U13
73
75
77
79 A7
U8 74193
U12
DM8095
63 A8
49
55
57
HI ADDR BUS
U7 74193
U11
DM8095
51
59
53
61 A15
MEMORY ADDRESS
7410
U19
7410
U19
U6 8212
DS2
DS1
STB
CLR
MD
35 33 31 29 27 25 23 21
D7 DATA BUS D0
U21
D25
R34 220
D26
R35 220
D27
R36 220
C6
C7
C8
C9
+5
MCP-80/B
SCALE: NONE
DATE: 11 MAY 75
APPROVED BY
DRAWN BY
Tychon, Inc. BLACKSBURG, VA 24060
DRAWING NUMBER
80003

E & L Instruments MD-1 8080-based microcomputer.


HI ADDRESS
LO ADDRESS
MEMORY DATA
+5
R10
D1
R17
D8
R18
D9
R25
D16
R26
D17
R33
D24
A15
A8
A7
A0
MD7
MD0
61
53
59
51
57
55
49
63
79
77
75
73
71
69
67
65
66
68
70
72
74
76
78
80
MCP-80/B LED REGISTER DISPLAY

NOTES

1. @ = U16, † = U18, # = U20, * = U22
2. PARTS IN SEQUENCE

MCP-80/B LED DISPLAY

SCALE: NONE
DATE: 11 MAY 75

Tychon, Inc. BLACKSBURG, VA 24060

DRAWING NUMBER 80004

Fig. 9-2. LED indicators for the hardwired front panel shown in Fig. 9-1.

ory location. If this is true, the N key is pressed so that the next consecutive memory address and its content will be displayed. The N key represents the *NEXT* command. If we want to change the value stored in this addressed memory location, we simply have to enter a three-digit octal (two-digit hexadecimal) number.

The simplest, but not necessarily the best, method of doing this would be to have the system monitor programmed so that an octal or hexadecimal number can be entered at any time. Once it is entered, it is automatically saved in the memory location addressed by the 16-bit address displayed on the LED displays. The memory address is then incremented, and then this new memory address and its content are displayed.

Suppose that we have to save 341 (E1) in memory location 005 135 (055D). To do this, you would press the A key, followed by 005 135, and then press 341. However, suppose you made a mistake as you entered the last digit of the 341 data value. What would happen if 342 were entered? The 342 would be saved in memory location 005 135 and the memory address would be incremented to 005 136. This new memory address and its content would then be displayed. To change the 342 in memory location 005 135, you would have to enter the 16-bit address (A 005 135) and then enter the 341. If another mistake were made while the 341 was being entered, the 16-bit address would have to be reentered along with the correct numeric value to be saved in that memory location.

As you can see, a tremendous amount of effort is required to correct mistakes. Therefore, it would be better to let the user enter as many octal or hexadecimal digits as desired. As each new digit is entered, the previous eight-bit number is shifted to the left, making room for the three-bit octal digit or four-bit hexadecimal digit. Only when the S key on the keyboard is pressed (representing the *SAVE* command), is the eight-bit number that is *displayed* actually saved in the specified memory location. When this "save" action is performed, the memory address is incremented and the new address and data values are displayed. Thus, when the S key is pressed, the memory address is automatically incremented; we do not have to press another key in order to increment the memory address.

The S key (SAVE command) can also be used to perform all of the operations that the N key (NEXT command) performed. This means that the S key can be used not only to save the "displayed" information in memory, but it can also be used to examine consecutive memory locations. A flowchart for this simple system monitor is shown in Fig. 9-3. The system monitor program that performs these operations is listed in Example 9-1.

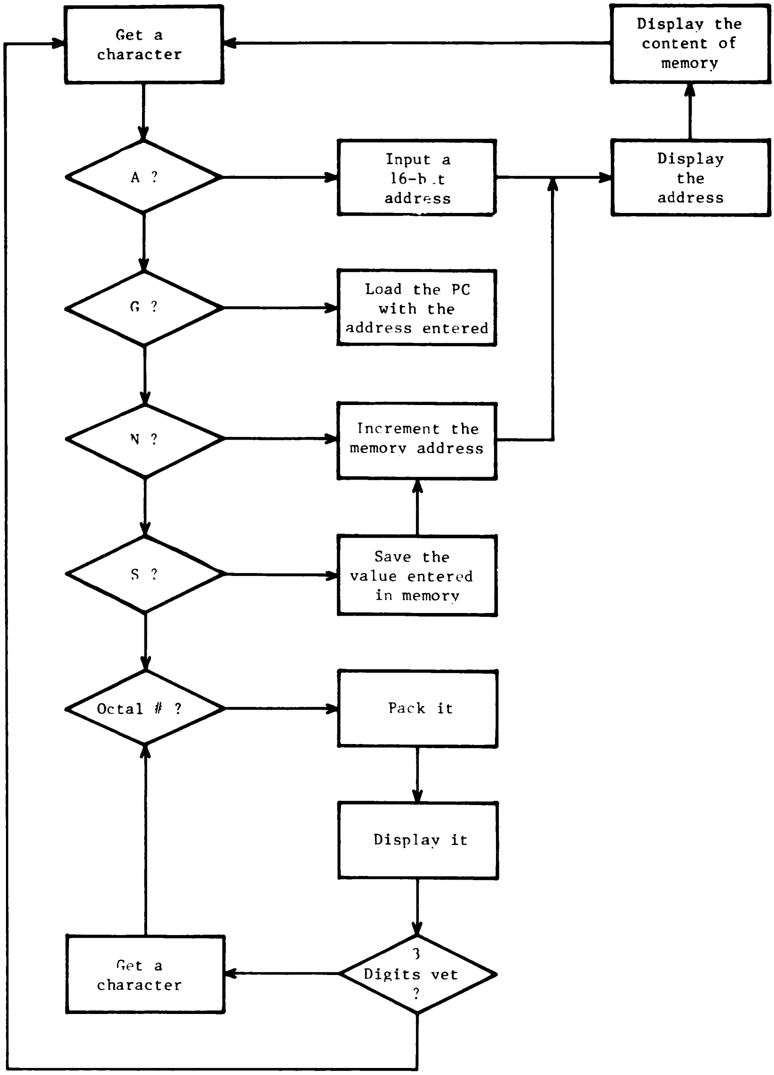

Get a character
Display the content of memory
A ?
Input a 16-bit address
Display the address
G ?
Load the PC with the address entered
N ?
Increment the memory address
S ?
Save the value entered in memory
Octal # ?
Pack it
Display it
Get a character
3 Digits yet ?

Fig. 9-3. Flowchart for a simple system monitor.

A SIMPLE SYSTEM MONITOR PROGRAM

The system monitor listed in Example 9-1 has four commands. These commands give us the ability to specify a 16-bit address (the A or *ADDRESS* command), to increment the memory address by 1 (the N or *NEXT* command), to save an eight-bit data value in a memory location (the S or *SAVE* command), and to begin

Example 9-1: A Simple System Monitor That Has Four Commands

```
        *000 000
START,  LXISP   /LOAD THE STACK POINTER WITH A R/W
        300     /MEMORY ADDRESS.
        003
CMD,    CALL    /GET AN ASCII CHARACTER
        KEYIN   /FROM THE KEYBOARD.
        0
        CPI     /WAS THE A FOR "ADDRESS" KEY PRESSED?
        101
        JZ      /YES, THEN A SIX-DIGIT ADDRESS
        ADDR    /MUST FOLLOW THE A.
        0
        CPI     /WAS THE G FOR "GO" KEY PRESSED?
        107
        JNZ     /NO, THE G KEY WAS NOT PRESSED,
        NORS    /SEE IF IT WAS THE S KEY.
        0
        PCHL    /IT WAS A G, USE H AND L AS AN ADDRESS.
NORS,   CPI     /WAS THE S KEY PRESSED?
        123
        JNZ     /NO, THE S KEY WAS NOT PRESSED, SO
        NOTS    /SEE IF THE N KEY WAS PRESSED.
        0
        MOVMD   /THE S KEY WAS PRESSED, SAVE THE VALUE
        JMP     /IN D IN MEMORY. INCREMENT THE
        ADRUP   /MEMORY ADDRESS AND DISPLAY THE NEW
        0       /ADDRESS AND THE CONTENT OF MEMORY.
NOTS,   CPI     /WAS THE N KEY PRESSED?
        116
        JNZ     /NO, THE N KEY WAS NOT PRESSED,
        NMBIN   /SO SEE IF AN OCTAL NUMBER IS
        0       /BEING ENTERED.
ADRUP,  INXH    /INCREMENT THE MEMORY ADDRESS AND
        JMP     /THEN DISPLAY THE NEW ADDRESS AND
        ADROUT  /THE CONTENT OF MEMORY AT THAT
        0       /ADDRESS.
NMBIN,  LXID
        003     /LOAD THE DIGIT COUNT INTO E
        000     /AND CLEAR D FOR TEMPORARY STORAGE.
        CALL    /THEN CALL A SECTION OF THE "CONVERSION"
        SKIP    /SUBROUTINE, BECAUSE THE A REGISTER ALREADY
        0       /CONTAINS THE ASCII VALUE FOR THE KEY.
        JMP     /JUMP BACK TO THE COMMAND DECODER
        CMD     /WITH THE VALUE ENTERED IN THE D
        0       /REGISTER.
OCTIN,  LXID    /LOAD THE E REGISTER WITH 003
        003     /AND THE D REGISTER WITH 000.
        000
IGNOR,  CALL    /GET  AN ASCII VALUE FROM
        KEYIN   /THE KEYBOARD.
        0
SKIP,   CPI     /IS IT LESS THAN ASCII 0?
        060
        JC      /YES, THEN IGNORE IT.
        IGNOR
        0
        CPI     /IS IT EQUAL TO OR GREATER THAN ASCII
        070     /8?
        JNC
        IGNOR   /YES, THEN IGNORE IT.
        0
        ANI     /IT'S AN OCTAL NUMBER, SO
        007     /PROCESS IT.
        MOVBA   /SAVE THE 3 LSB'S IN B.
        MOVAD   /GET THE PREVIOUS NUMBER
        RLC     /AND MULTIPLY IT BY EIGHT.
        RLC
        RLC
        ANI     /DISCARD THE 3 LSB'S.
        370
        ADDB    /ADD THE NUMBER JUST ENTERED.
        MOVDA
        OUT     /OUTPUT THE NUMBER TO THE
        002     /"DATA" SEVEN-SEGMENT DISPLAYS.
        DCRE    /DECREMENT THE DIGIT COUNT.
        JNZ     /IF THE COUNT IS NONZERO,
        IGNOR   /GET ANOTHER NUMERIC CHARACTER.
        0
        RET

ADDR,   CALL    /THE A KEY WAS PRESSED, SO GET A SIX-
        OCTIN   /DIGIT MEMORY ADDRESS.
        0
        MOVHD   /SAVE THE FIRST THREE DIGITS IN H.
        CALL    /THEN GET THE LO ADDRESS.
        OCTIN
        0
        MOVLD   /AND SAVE IT IN THE L REGISTER.
ADROUT, MOVAH   /THEN DISPLAY THE HI ADDRESS.
        OUT
        001
        MOVAL   /AND THE LO ADDRESS.
        OUT
        000
        MOVDM
        MOVAM   /GET THE CONTENT OF MEMORY AT THIS
        OUT     /ADDRESS AND DISPLAY IT ALSO.
        002
        JMP     /THEN GET ANOTHER COMMAND OR
        CMD     /DATA WORD.
        0
KEYIN,  IN      /INPUT THE STATUS WORD OF THE ASCII
        001     /KEYBOARD.
        ANI     /SAVE ONLY THE KEYBOARD'S FLAG IN THE
        200     /A REGISTER.
        JZ      /THE FLAG IS A LOGIC ZERO, SO NO
        KEYIN   /KEY IS PRESSED.
        0
        IN      /INPUT THE EIGHT-BIT ASCII VALUE AND
```

```
000     /CLEAR THE KEYBOARD FLAG.
ANI     /SET THE PARITY BIT OF THE ASCII
177     /VALUE TO ZERO.
RET     /RETURN WITH THE ASCII VALUE IN THE
        /A REGISTER.
```

executing a program that is stored in memory (the G or *GO* command). At *all times*, this system monitor uses a 16-bit address when information is saved in memory, when we examine the content of memory, or when the 8080 jumps to a specified memory location to begin the execution of a program.

After setting the stack pointer with a R/W memory address, the 8080 gets an ASCII value from the keyboard by calling the KEYIN subroutine. Some typical command decoder instructions are then executed. If the A key on the keyboard is pressed, the 8080 jumps to ADDR. At ADDR, the OCTIN subroutine is called twice so that a 16-bit address can be input in the form of two three-digit octal numbers. After the address is input, the 8080 outputs the content of register pair H and the content of the memory location addressed by register pair H to the LED displays (at ADROUT). When these operations have been performed, the 8080 jumps back to CMD, so that another command can be input and decoded.

If the G key has been pressed, representing the *GO* command, the displayed address, which is also contained in register pair H, is used as the starting address for program execution, and the program starting at this address is executed. The only way that program control will return to the system monitor is if the 8080 microcomputer is reset.

If the S key (*SAVE* command) has been pressed, the 8080 saves the content of the D register in the memory location addressed by register pair H. The 8080 then jumps to ADRUP, where the content of register pair H is incremented by 1. By jumping to ADROUT, the content of register pair H and the content of the memory location addressed by register pair H are displayed on the LED displays. The 8080 then jumps back to CMD so that another command can be entered.

If the N key (*NEXT* command) has been pressed, the content of register pair H is incremented by 1 and the incremented result is displayed, along with the content of the memory location addressed by register pair H. The 8080 then jumps back to CMD after these operations have been performed. If none of these four commands has been entered, the 8080 tries to interpret the ASCII value as part of an octal number. Therefore, the instructions at NMBIN are executed.

At NMBIN, register pair D is loaded with 000 003 (0003). This clears the D register to 0, so that it can be used to store the result of the ASCII-based octal-to-binary conversion. The E register is loaded with 3 because three ASCII-based octal values will be converted to a single eight-bit binary number. The 8080 then calls the SKIP subroutine, which assumes that the A register already contains an ASCII value from the keyboard, and that the D and E registers have been initialized. Actually, the SKIP subroutine is a portion of the longer subroutine called OCTIN (for *OCT*al *IN*put). If the subroutine is called, starting at OCTIN, the initial ASCII value contained in the A register will be ignored. By calling the SKIP subroutine, the 8080 can process the ASCII value in the A register, and then call the KEYIN subroutine twice, so that a total of three ASCII values are input. The end of the subroutine that contains OCTIN and SKIP contains steps that display the current content of the D register, the eight-bit octal value entered on three numeric keys.

Suppose that the nine-digit display displays the address 012 300, and the content of memory at this address is 203, which is also displayed. What will be displayed if you now press the 2 key on the ASCII keyboard? The address displayed will still be 012 300, but the data value will be 002. Since the 8080 is still executing the instructions within the OCTIN subroutine, two additional numeric keys must be pressed before the 8080 will return from the subroutine. Once the two additional keys are pressed, the 8080 returns. *The eight-bit value entered as three ASCII digits is not stored in memory location 012 300. It is only stored in the D register.* When the S key is pressed, the number stored in register D will be stored in the addressed memory location, 012 300.

What will happen if at some later time the 7 key is pressed three times? The value 377 is displayed. Likewise, if the 5 key is pressed three times, the value 155 is displayed. Remember, the display only displays an eight-bit number as three octal digits. Therefore, the numbers 777 and 555 are truncated to the eight-bit values 377 and 155, respectively.

What would happen if as soon as this system monitor program is started the S key were pressed? The S key would cause the content of the D register to be saved in the memory location addressed by register pair H. What would be the address of this memory location? There is no way of knowing. Therefore, if the S key were pressed as soon as the system monitor was started, the 8080 would write the content of the D register into the memory location addressed by register pair H. What is the simplest solution to this problem? The simplest solution would be to jump to ADROUT as soon as the stack pointer is set. This would cause the content of register pair H to be displayed, along with the content of the memory location addressed by register pair H. The software also copies the content of this memory location into the D register so that if the S key were pressed, the content of the D register would be saved back in the *same* memory location. The net effect is that nothing would be changed.

What does the G key do? When the G key is pressed, the content of register pair H is loaded into the program counter. This means that register pair H contains the address of the next instruction in memory that is to be executed by the 8080. If you load a program into the microcomputer starting at memory address 002 145, then you will probably want to begin execution of the program, starting at 002 145. To do this, you would enter A 002 145 G. Assuming that you only have 1024 words of R/W memory, with addresses between 000 000 and 003 377 (0000 through 03FF), what will happen if you enter A 200 127 G? The 8080 will "jump" to memory location 200 127, which is a nonexistent memory location. It will fetch the instruction op code 377 (FF) from this nonexistent memory location, since the data bus will "float to a logic 1." This is the op code for a RST7 instruction, so the 8080 will call the subroutine at 000 070 (0038). The "instruction" at this address is the data byte 070 of a CPI instruction in the OCTIN (SKIP) subroutine.

Needless to say, you must be careful when the GO command is used. Make sure that the address being displayed is really the address that you want the 8080 to jump to. Suppose an A 000 000 G is entered into the microcomputer. What happens? The 8080 jumps to memory address 000 000 and begins execution of the program stored there. Since this is the starting address of the system monitor program, the system monitor is simply restarted.

The system monitor program listed in Example 9-1 operates on 16-bit and eight-bit octal numbers. If the A key is pressed, a 16-bit (six-digit octal) number must be entered next. If data values are to be saved in memory, they must be entered as eight-bit (three-digit octal) numbers. The system monitor program can be greatly simplified if it only has to input and operate on eight-bit numbers. We can still enter a 16-bit address, only it now has to be entered as two eight-bit numbers. If an eight-bit number is entered and we want it to represent the eight MSBs of the 16-bit address, we press the H key, the *HI address command.* This eight-bit value is then moved to the H register, and the display is updated to reflect this change in address.

The low eight bits of address are changed by entering an eight-bit number and then pressing the L key, the *LO address command.* Once the proper 16-bit address is displayed, the S key is used to either (1) save new information in the addressed memory location and then increment the memory address, or (2) simply increment the memory address. If you examine Example 9-1 very closely, you will see that the NEXT command (the N key) is not needed. If a number is entered on the keyboard and the S key is pressed, the new value is saved in the memory location addressed by register pair H, and the memory address is incremented. If the correct value is already contained in memory, the S key simply has the net effect of incrementing the memory address. Actually, no matter when the S key is pressed, the content of the D register is always written into the memory location addressed by register pair H. However, as soon as an address is specified with the A command, the content of memory addressed by register pair H is *moved to the D register*. Therefore, if the S key is pressed and no new numeric information is keyed in, it appears as if the memory address is simply incremented. In reality, the content of the D register is saved in memory before the memory address is incremented. After the address is incremented, the value contained in the newly addressed memory location is moved to the D register, ready to be either stored back in the location, or it may be changed and be written back into the location. Again, this latter action has a net effect of incrementing the address. This is an important point, since the S command allows you to both examine a location and leave it unchanged or changed. Since the S command does all that the N command does, we only need four commands in the new, simplified system monitor listed in Example 9-2. These commands are H, L, S, and G.

In Example 9-2, the 8080 first sets the stack pointer and then displays the address in register pair H and the content of that memory location. Note that the content of memory is not only moved to the A register and output, it is also moved to the D register. At IGNOR, the 8080 calls the KEYIN subroutine so that an ASCII value can be input from the keyboard. The 8080 then executes some command decoder instructions to determine if the H, L, S, or G key is pressed.

If the H key is pressed, the content of the D register is moved to the H register, and the new 16-bit address and data value are then displayed. If the L key is pressed, the content of the D register is moved to the L register, and the new address and data value are displayed. If the S key is pressed, the content of the D register is saved in the memory location addressed by register pair H. The memory address is then incremented and the new address and data values are displayed. If the G key is pressed, the content of register pair H is loaded into the program counter (PC).

If none of these keys is pressed, the 8080 tries to interpret the ASCII value as an octal digit. If the ASCII value does not represent a valid octal digit, the 8080 jumps to IGNOR. If the value is valid, it is converted to the corresponding three-bit binary value

Example 9-2: A Simplified Four-Command System Monitor

```
        *000 000
START,  LXISP   /LOAD THE STACK POINTER WITH
        300     /A R/W MEMORY ADDRESS.
        003
DISPLA, MOVAH   /GET THE HI MEMORY ADDRESS
        OUT     /AND DISPLAY IT.
        001
        MOVAL   /GET THE LO MEMORY ADDRESS
        OUT     /AND DISPLAY IT.
        000
        MOVAM   /GET THE CONTENT OF MEMORY
        MOVDM   /AT THAT ADDRESS AND
DOUT,   OUT     /DISPLAY IT.
        002
IGNOR,  CALL    /GET AN ASCII CHARACTER.
        KEYIN
        0
        CPI
        110     /WAS THE H KEY PRESSED?
        JZ      /YES, THEN MOVE THE NUMBER ENTERED
        HI      /INTO THE HI REGISTER AND DISPLAY
        0       /THE NEW ADDRESS.
        CPI     /WAS THE L KEY PRESSED?
        114
        JZ      /YES, THEN MOVE THE NUMBER ENTERED
        LO      /INTO REGISTER L AND DISPLAY
        0       /THE NEW ADDRESS.
        CPI     /WAS THE S KEY PRESSED?
        123
        JZ      /YES, THEN SAVE THE NUMBER ENTERED
        STEP    /IN THE MEMORY LOCATION ADDRESSED BY
        0       /REGISTER PAIR H AND INCREMENT THE ADDRESS.
        CPI     /WAS THE G KEY PRESSED?
        107
        JZ      /YES, THEN JUMP TO THE MEMORY
        GO      /LOCATION ADDRESSED BY REGISTER
        0       /PAIR H.
        CPI     /IS THE VALUE LESS THAN ASCII 0?
        060
        JC      /YES, THEN IGNORE IT.
        IGNOR
        0
        CPI     /IS THE VALUE EQUAL TO OR GREATER
        070     /THAN ASCII 8?
        JNC     /YES, THEN IGNORE IT.
        IGNOR
        0
        ANI     /NO, SAVE THE OCTAL NUMBER IN B.
        007
        MOVBA
        MOVAD   /GET THE PREVIOUS VALUE
        RLC     /AND SHIFT IT TO MAKE ROOM
        RLC     /FOR A NEW THREE-BIT DIGIT.
        RLC
```

```
         ANI      /SAVE ONLY THE FIVE MSB'S.
         370
         ADDB     /ADD THE NUMBER JUST ENTERED
         MOVDA    /AND SAVE IT IN D.
         JMP      /THEN DISPLAY THE NEW NUMBER
         DOUT     /AND GET ANOTHER ASCII CHARACTER.
         0
HI,      MOVHD    /MOVE THE NUMBER ENTERED INTO
         JMP      /THE H REGISTER AND THEN DISPLAY
         DISPLA   /THE NEW ADDRESS.
         0
LO,      MOVLD    /MOVE THE NUMBER ENTERED INTO
         JMP      /THE L REGISTER AND THEN DISPLAY
         DISPLA   /THE NEW ADDRESS.
         0
STEP,    MOVMD    /SAVE THE VALUE ENTERED IN MEMORY.
         INXH     /INCREMENT THE MEMORY ADDRESS
         JMP      /AND DISPLAY THE NEW ADDRESS.
         DISPLA
         0
GO,      PCHL     /LOAD THE PC WITH REGISTER PAIR H.
KEYIN,   IN       /INPUT THE STATUS WORD OF THE ASCII
         001      /KEYBOARD.
         ANI      /SAVE ONLY THE KEYBOARD'S FLAG IN THE
         200      /A REGISTER.
         JZ       /THE FLAG IS A LOGIC ZERO, SO NO
         KEYIN    /KEY IS PRESSED.
         0
         IN       /INPUT THE EIGHT-BIT ASCII VALUE AND
         000      /CLEAR THE KEYBOARD FLAG.
         ANI      /SET THE PARITY BIT OF THE ASCII
         177      /VALUE TO ZERO.
         RET      /RETURN WITH THE ASCII VALUE IN THE
                  /A REGISTER.
```

and combined with the shifted content of the D register. This new value is displayed because the 8080 jumps to DOUT.

Note that in Example 9-2, there is no OCTIN subroutine although there are still instructions that perform an ASCII-based octal-to-binary conversion. Also, no digit counter is used when an ASCII value for an octal number is entered. In Example 9-1, an LXID instruction was executed so that the E register was loaded with 3 (the digit count) and the D register with 0. The S key in Example 9-2 serves two purposes. If a new octal number is entered, the S key will cause this number to be saved in the memory location addressed by register pair H. The address is then incremented and displayed. If no new value is entered, the value that was read from memory is written back into the same memory location. *The only reason that the S key can perform both of these functions is because the 8080 executes a MOVDM instruction each time it executes the instructions after DISPLA.* Really, the D register is used to "link" the entire system monitor together. The content of the D register can be used as the eight MSBs or LSBs in an address, or it can be saved in memory.

A SYSTEM MONITOR FOR USE WITH NON-ASCII KEYBOARDS

We have assumed that an ASCII-encoded keyboard is interfaced to the microcomputer, which we can use to enter numeric values and commands. For some microcomputer users, an ASCII keyboard may be too expensive, or unavailable. Instead, they have a 16- or 25-key keyboard that they have interfaced to their 8080 system. However, the keys do not produce the ASCII codes that the system monitor in Example 9-2 expects. Instead, the keys produce the codes listed in Table 9-2. Can the system monitor in Example 9-2 be altered so that the commands and numeric information can be entered into the microcomputer using such a keyboard? Yes, we simply need a look-up table to convert the key codes produced by the keyboard to ASCII values. The new KEYIN subroutine for the system monitor is listed in Example 9-3.

The look-up table in Example 9-3 is used to convert the key codes produced by the keyboard, to the ASCII values that the system monitor requires. This means that the KEYIN subroutine is the only section of the system monitor shown in Example 9-2 that has been modified. After the 8080 inputs the key code in Example 9-3, bits D_4 through D_7 are set to 0 by the ANI 017 instruction. Register pairs D and H are then saved on the stack, because they will be used in the calculation of the look-up table address.

After register pairs D and H are saved on the stack, the four-bit key code is moved from the A to the E register, and the D register is set to 0. Register pair H is then loaded with the *base address* of the look-up table. The four-bit key code in register pair D is then

Table 9-2. The Codes Generated by a 16-Key Keyboard

Key	Code Generated by the Keyboard
0	012
1	000
2	003
3	013
4	017
5	005
6	001
7	014
H	004
L	007
G	010
S	011

Example 9-3: A KEYIN Subroutine That Uses a Look-Up Table for Key Code Conversion

```
                   *000 341
000 341 333 KEYIN, IN        /INPUT THE STATUS WORD FROM THE
000 342 001        001       /ASCII KEYBOARD.
000 343 346        ANI       /SAVE ONLY THE KEYBOARD'S FLAG
000 344 200        200       /IN THE A REGISTER.
000 345 312        JZ        /THE FLAG IS A LOGIC ZERO, SO NO
000 346 341        KEYIN     /KEY IS PRESSED.
000 347 000        0
000 350 333        IN        /INPUT THE EIGHT-BIT ASCII VALUE AND
000 351 000        000       /CLEAR THE KEYBOARD FLAG.
000 352 346        ANI       /SAVE ONLY BITS D3, D2, D1, AND D0
000 353 017        017       /IN THE A REGISTER.
000 354 325        PUSHD     /SAVE REGISTER PAIR D.
000 355 345        PUSHH     /SAVE REGISTER PAIR H.
000 356 137        MOVEA     /MOVE THE KEY CODE TO E.
000 357 026        MVID      /SET THE D REGISTER TO 000.
000 360 000        000
000 361 041        LXIH      /LOAD REGISTER PAIR H WITH THE
000 362 371        TABLE     /STARTING ADDRESS OF THE LOOK-UP
000 363 000        0         /TABLE.
000 364 031        DADD      /ADD THE KEY CODE TO THE ADDRESS.
000 365 176        MOVAM     /GET THE ASCII VALUE FROM THE TABLE.
000 366 341        POPH      /RESTORE REGISTER PAIR H.
000 367 321        POPD      /RESTORE REGISTER PAIR D.
000 370 311        RET       /AND RETURN WITH THE ASCII VALUE
                             /IN THE A REGISTER.

000 371 061 TABLE, 061       /THE 1 KEY PRODUCES THE CODE 000.
000 372 066        066       /THE 6 KEY PRODUCES THE CODE 001.
000 373 377        377       /NO  KEY PRODUCES THE CODE 002.
000 374 062        062       /THE 2 KEY PRODUCES THE CODE 003.
000 375 110        110       /THE H KEY PRODUCES THE CODE 004.
000 376 065        065       /THE 5 KEY PRODUCES THE CODE 005.
000 377 377        377       /NO KEY PRODUCES THE CODE 006.
001 000 114        114       /THE L KEY PRODUCES THE CODE 007.
001 001 107        107       /THE G KEY PRODUCES THE CODE 010.
001 002 123        123       /THE S KEY PRODUCES THE CODE 011.
001 003 060        060       /THE 0 KEY PRODUCES THE CODE 012.
001 004 063        063       /THE 3 KEY PRODUCES THE CODE 013.
001 005 067        067       /THE 7 KEY PRODUCES THE CODE 014.
001 006 377        377       /NO KEY PRODUCES THE CODE 015.
001 007 377        377       /NO KEY PRODUCES THE CODE 016.
001 010 064        064       /THE 4 KEY PRODUCES THE CODE 017.
```

added to the base address of the table (DADD), and the ASCII value for the key that was pressed is moved from memory to the A register. Register pairs H and D are then popped off of the stack, and the 8080 returns with the ASCII value in the A register. The original KEYIN subroutine in Example 9-2 did just this.

The important point here is that the key code has been used as an *address*. Since each key has a unique code and, thus, points to a unique address, the equivalent ASCII codes are stored in the locations "addressed" by the keyboard keys. In Table 9-2, the code generated by the 4 key is 017. When this value is added to the base address of the table (000 371), the result is the address 001 010. The content of this memory location is the ASCII value for 4; 064. By using a look-up table, the key code is used to address the memory location that contains the appropriate ASCII value.

A SYSTEM MONITOR FOR USE WITH A MULTIPLEXED DISPLAY

Our final system monitor example will use a nine-digit multiplexed seven-segment LED display and a 12-key keyboard. The interface for the multiplexed display and an explanation of the display software can be found in Chapter 7 of *8080/8085 Software Design—Book 1.*[1] The keyboard that we will use produces four-bit key codes and has one status bit that indicates whether or not a key is pressed. This keyboard is also described in Chapter 7 of *8080/8085 Software Design—Book 1.*[1] Although this keyboard does not produce ASCII codes, the codes for the keys 0 through 7 are 0000 through 0111. Also, the keyboard status bit is *not* cleared when the key code is input, and the keys are not debounced. The status bit simply indicates when a key is pressed, and it will remain in the logic 1 state as long as a key is pressed. To debounce the key closures and openings, time delay instructions will have to be executed.

Although many changes are required to convert the system monitor listed in Example 9-2 to a system monitor that uses this type of keyboard and display, most of the changes deal with the display of data on the multiplexed display and the instructions required to sense a key closure, debounce it, and input the corresponding key code. The system monitor program listed in Example 9-4 does all of this.

In Example 9-4, the 8080 loads the stack pointer with a R/W memory address, and then loads register pair H with a R/W memory address and reads a data value from this memory location into the D register. At AGAIN, the DISPLA subroutine is called. The contents of the D, L, and H registers are displayed on the nine-digit multiplexed display when this subroutine is called. The numbers are displayed from right to left, so the content of the D register is displayed first, followed by the content of the L register, and then the content of the H register. When a particular register is being displayed in seven-segment octal format, bits D_2, D_1, and D_0 are displayed first, followed by D_5, D_4, and D_3, and lastly D_7 and D_6.

At DISPLA, the digit enable code (the code for the one digit in the display that will be turned on) is set to 0. This value is stored in the E register. The contents of the D, L, and H registers are

Example 9-4: A System Monitor That Uses a Multiplexed Display and a Non-ASCII, Nondebounced 12-Key Keyboard

```
START,  LXISP   /LOAD THE STACK POINTER WITH A R/W
        370     /MEMORY ADDRESS.
        003
        LXIH    /LOAD REGISTER PAIR H WITH A R/W
        000     /MEMORY ADDRESS.
        003
        MOVDM   /GET A VALUE FROM MEMORY INTO D.
AGAIN,  CALL    /DISPLAY THE CONTENTS OF THE D, H,
        DISPLA  /AND L REGISTERS ON THE MULTIPLEXED
        0       /DISPLAY.
        IN      /THEN SEE IF ANY KEYS ARE PRESSED.
        000
        ANI     /SAVE ONLY THE MSB.
        200
        JZ      /NO KEY IS PRESSED, SO DISPLAY
        AGAIN   /THE REGISTERS FOR ABOUT 13.5 MSEC
        0       /AND THEN CHECK THE KEYBOARD AGAIN.
        CALL    /A KEY IS PRESSED, SO USE THE DISPLAY
        DISPLA  /SUBROUTINE AS THE TIME DELAY TO
        0       /DEBOUNCE THE KEY CLOSURE.
        IN      /THEN INPUT THE KEY CODE AGAIN.
        000
        ANI     /SAVE THE FOUR LSB'S.
        017
        CPI
        014     /IS THE H KEY PRESSED?
        JZ      /YES, THEN MOVE THE NUMBER ENTERED
        HI      /INTO THE HI REGISTER AND DISPLAY
        0       /THE NEW ADDRESS.
        CPI     /WAS THE L KEY PRESSED?
        015
        JZ      /YES, THEN MOVE THE NUMBER ENTERED
        LO      /INTO REGISTER L AND DISPLAY
        0       /THE NEW ADDRESS.
        CPI     /WAS THE S KEY PRESSED?
        010
        JZ      /YES, THEN SAVE THE NUMBER ENTERED
        STEP    /IN THE MEMORY LOCATION ADDRESSED BY
        0       /REGISTER PAIR H AND INCREMENT THE ADDRESS.
        CPI     /WAS THE G KEY PRESSED?
        013
        JZ      /YES, THEN JUMP TO THE MEMORY
        GO      /LOCATION ADDRESSED BY REGISTER
        0       /PAIR H.
        CPI     /IS THE NUMBER GREATER
        010     /THAN THE CODE FOR THE SEVEN KEY?
        JNC     /YES, THEN IGNORE IT.
        RELESE
        0
        ANI     /NO, SAVE THE OCTAL NUMBER IN B.
        007
        MOVBA
        MOVAD   /GET THE PREVIOUS VALUE
        RLC     /AND MULTIPLY IT BY EIGHT.
        RLC
        RLC
        ANI     /SAVE ONLY THE FIVE MSB'S.
        370
        ADDB    /ADD THE NUMBER JUST ENTERED
        MOVDA   /AND SAVE IT IN D.
RELESE, CALL    /NOW THAT A KEY CLOSURE HAS BEEN
        DISPLA  /DETECTED, WAIT FOR THE KEY TO
        0       /BE RELEASED.
        IN      /INPUT THE KEYBOARD CODE AND
        000     /STATUS BIT AGAIN.
        ANI     /SAVE ONLY THE MSB.
        200
        JNZ     /A KEY IS STILL PRESSED, SO WAIT
        RELESE  /FOR IT TO BE RELEASED.
        0
        JMP     /THE KEY IS RELEASED, SO WAIT FOR
        AGAIN   /ANOTHER ONE TO BE PRESSED.
        0
HI,     MOVHD   /MOVE THE NUMBER ENTERED INTO
        JMP     /THE H REGISTER AND THEN DISPLAY
        NEWCON  /THE 16-BIT ADDRESS AND CONTENT OF
        0       /MEMORY AT THIS ADDRESS.
LO,     MOVLD   /MOVE THE NUMBER ENTERED INTO
        JMP     /THE L REGISTER AND THEN DISPLAY
        NEWCON  /THE 16-BIT ADDRESS AND CONTENT OF
        0       /MEMORY AT THIS ADDRESS.
STEP,   MOVMD   /SAVE THE VALUE ENTERED IN MEMORY.
        INXH    /INCREMENT THE MEMORY ADDRESS.
NEWCON, MOVDM   /GET THE CONTENT OF MEMORY INTO D.
        JMP     /THEN DISPLAY THIS VALUE AND
        RELESE  /THE 16-BIT ADDRESS WHILE WAITING FOR
        0       /THE KEY TO BE RELEASED.
GO,     PCHL    /LOAD THE PC WITH REGISTER PAIR H.
DISPLA, MVIE    /SET THE DIGIT ENABLE CODE TO 000.
        000
        MOVAD   /DISPLAY THE CONTENT OF D FIRST.
        CALL
        DIGIT
        0
        MOVAL   /THEN DISPLAY THE LO ADDRESS.
        CALL
        DIGIT
        0
        MOVAH   /FINALLY, DISPLAY THE HI ADDRESS.
DIGIT,  MOVBA   /SAVE THE WORD IN B.
        ANI
        007     /GET THE THREE LBS'S INTO THE FOUR
        RRC     /MSB'S.
        RRC
        RRC
        RRC
        CALL    /DISPLAY THIS NUMBER.
        DIGOUT
        0
        MOVAB   /NOW DISPLAY THE INFORMATION
        ANI     /IN BITS D5, D4, AND D3.
        070
        RLC     /ROTATE IT ONCE LEFT.
        CALL    /DISPLAY THIS NUMBER.
        DIGOUT
        0
        MOVAB   /NOW GET BITS D7 AND D6.
        ANI
        300
        RRC     /ROTATE THIS INFORMATION TWICE
        RRC     /TO THE RIGHT.
DIGOUT, ADDE    /ADD THE DIGIT ENABLE CODE.
        OUT     /OUTPUT THE VALUE TO THE
        000     /DISPLAYS.
        INRE    /INCREMENT THE DIGIT ENABLE CODE.
        MVIC    /NOW INTENSIFY THE DISPLAY.
        310
INTENS, DCRC
        JNZ
        INTENS
        0
        RET
```

then loaded into the A register, one at a time. After each is loaded, the DIGIT subroutine is called to display the information of each register on the seven-segment display. Note that after the MOVAH instruction has been executed, the 8080 will continue on into the DIGIT subroutine. This means that the RET instruction at the end of the DIGIT subroutine also causes the 8080 to return to the section of the program that called the DISPLA subroutine.

At DIGIT, the content of the A register is saved in the B register. The 8080 rotates the bits to be displayed into bits D_6, D_5, and D_4 of the A register. For bits D_2, D_1, and D_0 to be displayed, this means that four RRC instructions must be executed. To display bits D_5, D_4, and D_3, only one RLC must be executed. Once the information to be displayed has been positioned into bits D_6, D_5, and D_4, the DIGOUT section of DISPLA is executed. At DIGOUT, the digit enable code contained in the E register, which lights only one digit in the display, is added to the information contained in the A register. This information is then output to the display interface. Bits D_6, D_5, and D_4 contain the value to be displayed, and bits D_3, D_2, D_1, and D_0 determine the digit in the display that will display the information contained in the four MSBs. The 8080 then executes a time delay loop at the end of DIGOUT, so that the individual displays are turned on for about 1.5 ms. After the 8080 has displayed the contents of the D, L, and H registers in this manner, it returns from the subroutine to the IN instruction just after AGAIN. The total time required to display all nine digits of information is about 13.5 ms.

The IN instruction just after AGAIN inputs the status bit and the data bits from the keyboard into the A register. The next two instructions determine whether or not a key in the keyboard is pressed. If no key is pressed, bit D_7 is a logic 0, so the JZ to AGAIN is executed. This means that the 8080 displays the contents of the D, L, and H registers again and, 13.5 ms later, executes the same IN instruction. Thus, the 8080 checks the keyboard every 13.5 ms to see if a key is pressed. If a key is pressed, bit D_7 of the A register is a logic 1, so the 8080 calls the DISPLA subroutine. This not only displays the data again, but it generates a time delay that is sufficiently long to "debounce" the key closure of the key that is pressed. After this "debounce software" has been executed, the 8080 inputs the key code into the A register.

Command decoder instructions are then executed to determine if the H, L, S, G, or a numeric key is pressed. Note that we have to know the codes produced by the keyboard for *all* of the keys in order to program the 8080 with the correct immediate data bytes for the CPI instructions. The operations that the 8080 performs if the H, L, S, or G keys are pressed have already been discussed. In this regard, this example is very similar to Example 9-2. Once the H, L, or S command has been processed, the 8080 jumps to NEWCON. At NEWCON, the content of the memory location addressed by register pair H is moved to the D register, and the 8080 jumps to RELESE.

If a numeric key has been pressed, rather than the H, L, S, or G key, the 8080 performs a number-base conversion and converts the key codes entered to eight-bit binary numbers. The result of this conversion is saved in the D register. Once the numeric information has been processed, or the 8080 has performed the operations required by one of the commands, it executes the instructions at RELESE.

From the time that the 8080 detected that a key was initially pressed, only 50 to 100 μs were required to process the key code. Therefore, by the time the 8080 gets to RELESE, *the user still has his finger on the key*. The 8080 must, therefore, wait for the user to release the key before doing anything else, so the DISPLA subroutine is called. This causes the contents of the D, L, and H registers to be displayed again. Every 13.5 ms, the 8080 checks to see if the key has been released. If it has not been, the DISPLA subroutine is called again. If the key has been released, the 8080 jumps to AGAIN, where the DISPLA subroutine is called. Thus, the key bounce caused by the key being released is ignored by the system monitor software.

What modifications would have to be made to Example 9-4 if the keys bounce for 20 ms? The time required to display the individual digits can be increased or decreased by changing the immediate data byte for the MVIE instruction at the end of the DISPLA subroutine. The maximum delay that can be produced by this section of the subroutine is about 2 ms. Therefore, the nine digits of information will only require 18 ms to be displayed, which is not long enough to debounce the key bounce. The solution to this problem is *not* to change the end of the DISPLA subroutine so that a larger number is either incremented or decremented. But, suppose that this was done and each digit required 5 ms to be displayed. This means that the nine digits would be displayed in 45 ms. The result is that the entire number would be displayed only 20 times every second. The display would be difficult to read because it would flicker.

The solution to the problem is to leave the DISPLA subroutine the way it is. However, instead of calling the DISPLA subroutine once at AGAIN, once at RELESE, and once to debounce the key closure, the DISPLA subroutine should be called twice in each place. This means that the debounce delay would be 27 ms rather than 13.5 ms. Since there are three calls to DISPLA, only three additional call instructions (nine memory locations) must be used to generate a debounce delay of 27 ms.

LINKING PROGRAMS WITH A SYSTEM MONITOR

So far, all of the system monitor examples that we have discussed have two characteristics in common. They all have the same type of command decoder, and they have some type of number-base conversion subroutine or instruction sequence. However, not all system monitor programs provide you with the ability to examine and/or modify the contents of memory. In fact, a system monitor can be written so that it is *just a command decoder.* This type of system monitor can be thought of as a hub in a wheel, linking together all the other programs stored in the microcomputer system, generally in ROM or PROM (Fig. 9-4).

In Fig. 9-4, you can see that the system monitor will always be started when the 8080 is reset. By entering various program names into the microcomputer system, we can execute one of the programs that is stored in memory, along with the system monitor. This means that in simple microcomputer systems, the system monitor and all the other programs are stored in memory *at the same time.* One of the methods often used to do this is to store all of the programs in ROM. Therefore, when power is applied to the microcomputer system, a number of programs are available for immediate use. The programs could be stored in R/W memory, but this means that

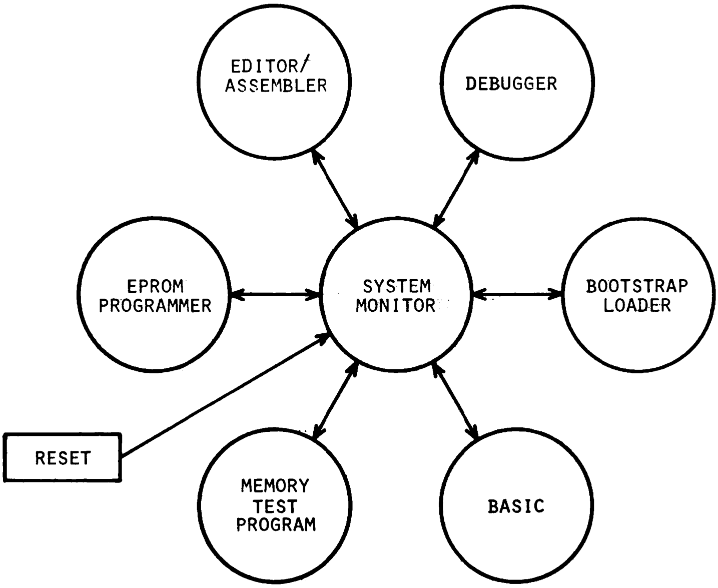


Fig. 9-4. Using a system monitor to link other programs together.

they would have to be loaded into the microcomputer system each time the power is turned on.

The system monitor that "links" all of these programs together can be a very simple or a very complex command decoder. Once a program name is entered into the microcomputer system using a teletypewriter or crt, the 8080 tries to match this name with one of the names in the command/command-address list. If a match occurs, the 8080 jumps to the starting address of the program (which is contained in the list used by the command decoder). Once this program has performed all of its required tasks, there has to be some method of returning to the system monitor. Based on the data in Fig. 9-4, the simplest way to return to the system monitor would be to reset the 8080. However, a command is often incorporated in all of the other programs, so that it is very easy to jump back to the system monitor.

The list contained within the system monitor can be used for two purposes. Any command entered is compared to the entries in the list. If a match occurs, the 8080 fetches a 16-bit address from the list and jumps to this address. The list can also be used by the 8080 to print a *directory* (the term *menu* is sometimes used) of the programs that will be recognized by the command decoder software steps within the system monitor. This is demonstrated in Example 9-5. Printing the directory requires a little effort because the list used not only contains the ASCII characters that make up the commands, but also an eight-bit character count and the 16-bit starting address of each program.

Example 9-5: The Output of the System Monitor Command List

```
PROGRAMS ON FILE:
BOOT
DBUG
DBUGII
DUP
LANDM
LETTER
TEA
UPP
VER

ENTER A COMMAND, THEN A CR.
COMMAND =
```

Based on Example 9-5, the first operation that the microcomputer must perform is to print the message "PROGRAMS ON FILE:", since this is not included in the command/command-address list used by the command decoder contained within the system monitor. The software required to print this message is very simple and has been used in previous examples, such as Example 5-14, where the NXTLET subroutine was used to print the message "TEST ZIP CODE." After this message is printed, the 8080 must print the commands (program names) stored in the command/command-address list.

To do this, the 8080 has to read the number of characters for the command into a register and then print this number of characters on the teletypewriter or crt. The 8080 must then skip over the 16-bit address stored after the ASCII characters and then print a carriage return and line feed (which are not contained in the list). By printing the carriage return and line feed, each program name will be printed on a separate line. When the 8080 reads a character count of 0 from the list, it knows that it has printed out all of the program names.

The 8080 then prints the message "ENTER A COMMAND, THEN A CR.", followed by a carriage return and line feed. The message continues with "COMMAND =". The command decoder instructions within the system monitor are then executed so that a multiple-letter command can be entered on the teletypewriter or crt and stored in R/W memory. Once the command is entered and terminated with a carriage return, the 8080 determines if it is a valid command. If it is, the 8080 reads a 16-bit address from the list and jumps to this address. If the command is not valid, the 8080 repeats the message "ENTER A COMMAND, THEN A CR.", and "COMMAND =", so that another command can be entered.

There are a number of operations that system monitors can perform other than the ones that we have discussed. For instance, system monitors can be used to directly communicate with input and output ports. Others can be used to enable and disable peripheral devices, so that during one minute a program is "talking" with a line printer, and during the next minute it is "talking" with a floppy disk. The program does not have to know that these two peripheral devices have been switched in and out. System monitors can also be used to transfer data from one peripheral to another or to monitor the operations of an applications program.

REFERENCE

1. Titus, C. A., Rony, P. R., Larsen, D. G., and Titus, J. A. *8080/8085 Software Design—Book 1.* Howard W. Sams & Co., Inc., Indianapolis, IN, 1978.

10

Breakpoints and Debuggers

In the previous chapter, we discussed system monitors and a number of features that they have. As you know, the system monitors that we have described can be used to load a program into memory, examine it and, if required, modify it. Once a complete program has been loaded into memory, the system monitor can be used to transfer program execution from the system monitor to a program in memory. What happens if the program that you entered into memory does not give you the expected results? This may mean that a teletypewriter does not print the correct message, the result of a mathematical calculation is incorrect, or that a stepper motor is not correctly controlled by the microcomputer.

If we do have problems with a program, the only action that we can take is to examine our program with a very critical eye and see if we can locate any mistakes (*bugs*). If a *bug* is located, we can change some instructions or data values in the program, using the system monitor, and then execute the program again. Unfortunately, we may have to execute the program 10 or 15 times in order to locate all of the bugs. If the program has to be executed for five or 10 minutes each time, then *debugging* the program will require a considerable amount of time. If no bugs can be located by examining the program, we may have to write some programs that test various peripheral devices or subroutines within the main program.

Because this method of debugging a program can be very time consuming, a *debugger* program is often used to help us locate errors in the program. There are debuggers for assembly language, BASIC, and FORTRAN programs. In fact, it is safe to say that there is some type of debugger for every computer language that has been written. A command summary for a typical 8080 assembly language debugger is shown in Table 10-1.

As you can see from Table 10-1, the debugger program is a teletypewriter-oriented program that has the ability to perform many of the operations of the system monitors described in the previous chapter. To examine the content of a memory location, a six-digit octal memory address (XXX YYY) is entered into the microcomputer, followed by a slash (/). The content of the addressed memory location is then printed immediately after the slash. To change the content of this memory location, we simply have to enter a valid three-digit octal number, and then press the line-feed key (LF). This will cause the new octal value to be saved in memory. It will also cause the next consecutive memory address to be printed,

Table 10-1. A Command Summary for a Typical 8080 Debugger[1]

1. XXX YYY / OLD	Output the content (OLD) of memory location XXX YYY.
2. XXX YYY / OLD NEW **LF** XXX YYZ / ABC	Save the new data word (NEW) in memory location XXX YYY and then output the content of the next consecutive memory location (XXX YYZ).
3. XXX YYY / OLD **CR**	Return to the command mode of the debugger after seeing the content of memory location XXX YYY.
4. XXX YYY L	List the contents of memory, starting at the specified memory address (XXX YYY).
5. XXX YYY G	Begin executing the program stored in memory starting at XXX YYY.
6. P XXX YYY AAA BBB	Punch the contents of memory locations XXX YYY through AAA BBB on paper tape.
7. R	Read the contents of a paper tape into memory.
8. XXX YYY B	Set a breakpoint at memory address XXX YYY.
9. K	Remove the **last** breakpoint from the program.
10. S	Execute a single instruction and print the state of the flags, and the contents of the registers and the stack on the teletypewriter.
11. X	Continue executing the program at full speed, starting at the last breakpoint address.

followed by the content of this memory location. If the carriage-return key (CR) is pressed after the content of memory is printed, rather than the line-feed key, the 8080 will return to the command mode of the debugger, so that a new command or memory address can be entered.

Since a teletypewriter is used to communicate with the microcomputer, a *list* feature was also incorporated into the debugger. This means that the contents of a number of consecutive memory locations can be listed on the teletypewriter without our intervention. The debugger also has a GO command. After entering a six-digit octal number, the G key of the teletypewriter is pressed. The 8080 then jumps to this address and begins to execute the program at this address.

The debugger also has the ability to save the contents of memory by having it punched on paper tape. This means that we can write a program, enter it into memory, debug it, and save it on paper tape for future use. At some later time, the debugger can be used to read the paper tape back into memory. The ability to save programs on some type of storage device is very important, although paper tape itself does not have to be used. In many microcomputer systems, audio cassettes and "floppy" disks are used as mass storage devices.

BREAKPOINTS

The last debugger feature that we will discuss, and one of the features that distinguishes a debugger from a system monitor, is the *breakpoint* feature. The debugger can set a *breakpoint* in a program that needs to be debugged. A breakpoint is one method that we can use to stop the execution of a program at a *specific instruction*. Program control (execution) is then returned to the debugger. At this point, the debugger can be used to examine or modify the contents of memory, the contents of the 8080 general-purpose registers, or instruction op codes. A breakpoint may be composed of hardware, software, or both. We will only discuss the software methods of implementing a breakpoint.

Quite often, a bug results because we are improperly using a register or memory location. A program may be storing two eight-bit bytes in the same register, or the program may be saving a 16-bit value in two nonconsecutive memory locations when two consecutive memory locations should have been used. To find these errors in the program, we will have to observe the contents of a register or registers, or possibly the contents of a series of memory locations, after a particular instruction or series of instructions have been executed.

This means that at some point in our program we must set a breakpoint. We then instruct the 8080 to begin executing our program at its starting address. The 8080 then executes the instructions in the program until the breakpoint is reached. Program control is then transferred back to the debugger. At this point, the contents of the 8080 general-purpose registers may be printed on the teletypewriter. This will tell us what the *state* of the 8080 was when the breakpoint was reached. By comparing the contents of the registers, or the contents of some memory locations, to values that we expect to be present when the program is operating properly, we can determine if there is a bug in the program between the beginning of the program and the address of the breakpoint. If the values in the 8080 registers are equal to the values that we expect, then we have either (1) not reached the bug in the program yet or (2) passed the bug many "instructions back." By moving the breakpoint back and forth in the program and restarting the program from the beginning we can "home-in" on the bug—the instruction or sequence of instructions that is causing the error.

Note that the debugger cannot *find* the bug, simply because the 8080 does not know what results the instructions stored in its memory are supposed to produce. Instead, the debugger can be used to show us the state of the microcomputer at a particular point in a program. It is our responsibility to determine whether or not the state of the 8080 is correct. If some of the values in the general-purpose registers are not correct, then the breakpoint must be moved back a few instructions and the program reexecuted. By moving the breakpoint around in the program, and by observing the state of the 8080 at the different breakpoint positions, we should be able to find the bug. If we find that the state of the 8080 is correct at a particular breakpoint position, then we need to move the breakpoint ahead one or more instructions. Rather than restart the program from the beginning, it would be advantageous if we could instruct the 8080 to *continue* program execution from the last breakpoint to the point where the next breakpoint is set. Of course, for the continue command to be useful, the state of the 8080 must be restored to what it was when the first breakpoint was reached before it continues on with the excution of the program.

As you can see, the debugger has to perform a number of complex operations in order to have a breakpoint feature. Some of the characteristics of a breakpoint feature that we will discuss include

1. Suitable breakpoint instructions.
2. Manually setting and clearing the breakpoint.
3. Automatically setting and clearing the breakpoint.
4. Saving and printing the contents of the registers when the breakpoint is reached.

5. Breakpoint operation.
6. The requirements for a continued command.
7. Single-stepping—executing one instruction at a time—under control of the debugger.

BREAKPOINT INSTRUCTIONS

How can we actually cause program control to return to the debugger when the breakpoint is reached? The simplest solution is to *insert* an instruction into the program being debugged. This instruction, when executed, will transfer control back to the debugger. The 8080 transfer-of-control instructions—jumps, calls, and restarts—do this quite well. To insert such an instruction into a program, the debugger could write a jump, call, or restart instruction *over* the instruction contained at the point from which we want to "break away" from the program being executed, back to the debugger. Therefore, this point at which control is returned from the program being debugged, to the debugger, is called the *breakpoint.* The address of this point is often referred to as the *breakpoint address.*

To place a jump or call instruction in the program being debugged, three eight-bit bytes would have to be written into the program. The op code for the jump or call instruction would have to be written into memory at the breakpoint address, and the second and third bytes (address bytes) of the instruction would have to be written into the next two consecutive memory locations. The restart instructions are single-byte instructions, so only a single byte would have to be written into the program being debugged if it were used. There is an advantage in using a three-byte jump or call instruction as the *breakpoint instruction* used to return control to the debugger program. What is this advantage? When a restart instruction is executed, the 8080 always calls a subroutine within addresses 000 000 through 000 070 (0000 through 0038). Since jump and call instructions are three-byte instructions, any address may be specified. Therefore, if a jump or call is used as the breakpoint instruction, we can immediately and conveniently "get back to" the debugger. If a restart instruction is used, the 8080 calls one of the subroutines within the first 56 memory locations and, starting at one of these memory locations (depending on the restart instruction used), we would need to have a jump instruction so that control would be returned to the debugger. Even though it is easier to insert a single-byte restart instruction into the program being debugged, a three-byte jump instruction would also be required, starting at the vector address of the restart instruction. For more information on restart instructions, we refer you to Chapter 4 of *8080/8085 Software Design—Book 1.*[2]

Once control is returned to the debugger, we would probably want to look at the contents of the 8080 registers when the breakpoint was reached in the program being debugged. In fact, it would be very difficult to debug a program without being able to examine the contents of the general-purpose registers. The contents of the general-purpose registers allow us to determine what the microcomputer was doing when it reached the breakpoint. We could determine what might have caused the "wrong" values to be in various registers. You might not even reach the breakpoint in the program being debugged. This would tell you something, too. Some error must have occurred between the start of the program and the breakpoint. This helps you to "narrow down" potential problems in your software.

There are basically two methods that can be used to examine the contents of these registers. The first method consists of printing out the contents of all of the registers each time the breakpoint is reached (the breakpoint instruction is executed). The second method leaves it up to the user to specify the registers to be printed, once the breakpoint is reached. Personally, we prefer to print out the contents of all the registers. By doing this, you have much of the information required to debug a program. Also, less user interaction is required because the user does not have to specify which registers are to be printed. If the contents of the registers are printed on a hard-copy device, such as a teletypewriter, then the user can go back and look at the printout and not have to wonder what was contained in the other registers.

Regardless of the method used, once program control is transferred to the debugger, it is a simple matter to call a subroutine so that the contents of the registers are printed in an easy-to-understand form. However, subroutines like this probably use three or four registers to transfer and print the register data on the teletypewriter or crt. Therefore, how can the contents of all the registers be stored so that the format-and-output subroutines within the debugger do not change these values? The simplest solution is to push all of the registers onto the stack as soon as possible after the breakpoint instruction has been executed. Once these values have been pushed onto the stack, we can pop them off as required and print their octal or hexadecimal equivalents on the teletypewriter or crt.

Now that we have described the operations that need to be performed, and the various instructions that can be used as the breakpoint instruction, let us look at the programs and subroutines that can actually be used.

MANUALLY SETTING AND CLEARING THE BREAKPOINT

Setting and clearing a breakpoint in a program can be a relatively simple process. In fact, you could use one of the system monitor programs listed in the previous chapter to do just this. After determining the address at which you want to "insert" a breakpoint, you would make a note on paper of the instruction op code contained in memory at this address, and also the contents of the next two consecutive memory locations. Then, starting at the breakpoint address, you would place a jump instruction op code (303, C3), followed by a 16-bit address, in the available R/W memory locations. The 16-bit address in the second and third bytes of the instruction would be the address at which the push instructions are stored that will save the contents of the registers on the stack when the breakpoint is reached. After the push instructions, there are steps that will print the contents of the registers on the teletypewriter or crt.

Once the breakpoint is "set" through the use of the jump instruction, you can begin execution of the program being debugged by entering a starting address into the 8080 using the system monitor. When the breakpoint is reached, the 8080 will jump to the instructions that save, and then print, the contents of the registers. After doing this, control will probably be returned to the system monitor so that the contents of memory can be examined or changed.

One improvement to this process, as mentioned previously, would consist of using a single-byte restart instruction rather than a three-byte jump instruction as the breakpoint instruction. This means that when the restart instruction is written into memory at the breakpoint address, only a single byte has to be written down on paper. After the breakpoint is reached and the registers are printed out, the system monitor can be used to place the original instruction op code back in the program at the breakpoint address. Of course, before the program is actually executed (using a restart instruction at the breakpoint address), a jump instruction has to be stored in memory at the vector address for the particular restart instruction that is used as the breakpoint instruction. This is illustrated in Fig. 10-1.

Note that if a RST3 instruction is used as the breakpoint instruction in Fig. 10-1, a jump instruction has to be stored in memory locations 000 030, 000 031, and 000 032 (0018, 0019, and 001A). This jump instruction transfers control back to the debugger when the RST3 instruction is executed. If a different restart instruction is used as the breakpoint instruction, *then a jump instruction must be placed at the appropriate vector address for that restart instruction.*

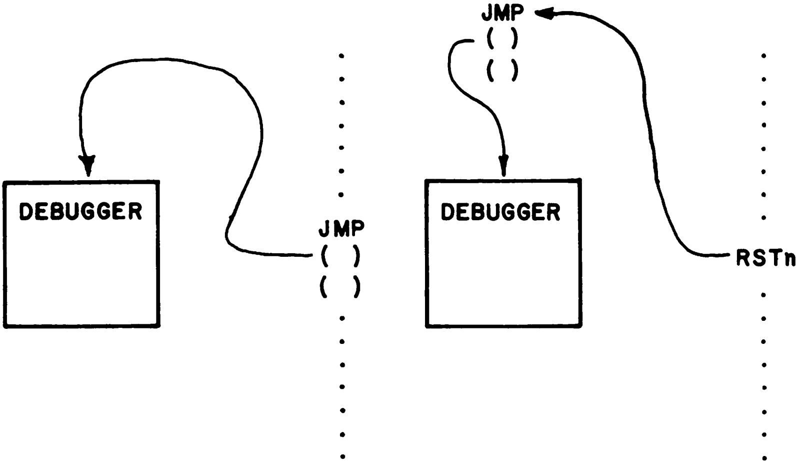


Fig. 10-1. Using a JMP or restart instruction for the breakpoint instruction.

The manual method of setting and clearing breakpoints that we have just discussed requires a considerable amount of effort; i.e., remembering the breakpoint address, and the instruction op code that was stored at this address, before the breakpoint instruction wrote over it. However, this method will work. One obvious improvement to the debugger would be to have the microcomputer *set* and *clear* the breakpoint *automatically* at the breakpoint address specified by the user.

AUTOMATICALLY SETTING AND CLEARING THE BREAKPOINT

Setting the Breakpoint

For the debugger to set the breakpoint, it should store a restart instruction in the program being debugged. Once the breakpoint is reached, the microcomputer should *clear* or "remove" the breakpoint by writing the original instruction back into the program over the restart instruction. To actually specify the 16-bit address for the breakpoint, you might have to enter into the microcomputer something like 004 023 B, BREAK AT 0413, or BRK 0413. Regardless of the method or format used to specify the use of the breakpoint, we will assume that the BREAK subroutine within the debugger is called when a breakpoint must be set in a program. Within this subroutine, the 8080 has to obtain the 16-bit breakpoint address from the user. This could be accomplished by calling an ASCII-based octal- or hexadecimal-to-binary conversion subroutine. Such ASCII-based octal- and hexadecimal-to-binary conversion subroutines are listed in Examples 6-3 and 6-9 in *8080/8085 Software Design—Book 1.*[2] These particular subroutines would have to be called *twice,* so that two eight-bit numbers could be entered into the microcomputer. To use one of these subroutines to enter a 16-bit breakpoint address, it could be called as shown in Example 10-1.

Example 10-1: Using the OCTIN Subroutine To Input a 16-Bit Address

```
BREAK,  CALL    /GET THE HI ADDRESS WHERE THE
        OCTIN   /BREAKPOINT IS TO BE SET.
        0       /(RETURN WITH THE NUMBER IN D.)
        MOVHD   /SAVE THE NUMBER IN H.
        CALL    /THEN CALL THE OCTIN SUBROUTINE SO THAT
        OCTIN   /THE LO ADDRESS OF THE BREAKPOINT
        0       /CAN BE ENTERED.
        MOVLD   /SAVE THIS NUMBER IN L.
         •      /THEN SET THE BREAKPOINT AT
         •      /THIS ADDRESS.
         •
```

We have assumed in Example 10-1 that the HI eight bits of the breakpoint address are entered into the microcomputer first, followed by the LO eight bits of the breakpoint address. Note that if the HEXBIN subroutine is used (Example 6-9 in *8080/8085 Software Design—Book 1*[2]), MOVHA and MOVLA instructions will have to be used in Example 10-1 rather than MOVHD and MOVLD

Example 10-2: Using the 8080 To Set the Breakpoint

```
BREAK,  CALL    /GET THE HI ADDRESS WHERE THE
        OCTIN   /BREAKPOINT IS TO BE SET.
        0       /(RETURN WITH THE NUMBER IN D.)
        MOVHD   /SAVE THE NUMBER IN H.
        CALL    /THEN CALL THE OCTIN SUBROUTINE SO THAT
        OCTIN   /THE LO ADDRESS OF THE BREAKPOINT
        0       /CAN BE ENTERED.
        MOVLD   /SAVE THIS NUMBER IN L.
        CALL    /THEN SET THE BREAKPOINT AT THE
        SETBRK  /MEMORY LOCATION ADDRESSED BY REG-
        0       /ISTER PAIR H.
        JMP     /ONCE THE BREAKPOINT IS SET, JUMP
        CMDDEC  /BACK TO THE COMMAND DECODER OF
        0       /THE DEBUGGER.
SETBRK, MOVAM   /GET THE "ORIGINAL" INSTRUCTION
        STA     /AND SAVE IT IN R/W MEMORY.
        ORGIN
        0
        MVIM    /THEN WRITE A RESTART INSTRUCTION OVER
        357     /THIS INSTRUCTION IN THE PROGRAM.
        SHLD    /SAVE THE ADDRESS WHERE THE BREAK-
        TEMPO   /POINT IS BEING SET IN
        0       /"TEMPO."
        RET     /THEN RETURN FROM SETBRK.
```

instructions. Remember, the address entered is the address at which you want to insert the breakpoint in the program being debugged. Since a restart instruction is to be "inserted" into the program, the original instruction in the program must first be *read* from memory and *saved* somewhere else in memory (scratchpad storage). Then the restart instruction can be written into the program being debugged. Using Example 10-1 as a starting point, we can now write a subroutine that sets the breakpoint and saves the original instruction that was replaced in the program being debugged. This new subroutine is listed in Example 10-2.

In Example 10-2, the 16-bit breakpoint address is entered into the microcomputer, via a teletypewriter or crt, in the form of two three-digit octal numbers. This address is saved in register pair H. The SETBRK subroutine is then called so that the content of the memory location addressed by register pair H is read from memory into the A register. This is the instruction that is stored at the breakpoint address in the program being debugged. This eight-bit instruction op code is then stored in ORGIN (*OR*i*G*inal *IN*struction; a R/W memory location). A RST5 instruction (357, EF) is then written into the memory location addressed by register pair H. This has the effect of writing a RST5 into the program being debugged.

Of course, if interrupt hardware (see Chapters 3 and 4) already uses the RST5 instruction, then a different restart instruction must be used for the breakpoint instruction. This means that the immediate data byte of the MVIM instruction in the SETBRK subroutine (Example 10-2) must be changed to the *op code* of one of the other unused restart instructions. Likewise, since we have to *write* a restart instruction into the program being debugged, *we cannot use this breakpoint technique to debug programs that are stored in read-only memory (ROMs), including a number of different classes of ROMs, such as PROMs and EPROMs.*

Let us mention an important point here. *You can only use a breakpoint at an address occupied by an instruction op code.* You must not place a breakpoint at an address occupied by data or address values. The breakpoint only substitutes one op code for another and, if you substitute the breakpoint op code for data or address values, the program will continue executing, interpreting the breakpoint op code as a new data or address byte.

Are there any additional operations that must be performed if a restart instruction is used as the breakpoint instruction? Yes, we must store a jump instruction at the vector address of the restart instruction so that control is transferred to the breakpoint portion of the debugger when the restart instruction is executed. Example 10-3 includes instructions that write a three-byte jump instruction (JMP) at the RST5 vector address. This assumes that adresses 000

Example 10-3: Setting the Breakpoint and Writing a JMP Instruction at the RST5 Vector Address

```
BREAK,   CALL    /GET THE HI ADDRESS WHERE THE
         OCTIN   /BREAKPOINT IS TO BE SET.
         0       /(RETURN WITH THE NUMBER IN D.)
         MOVHD   /SAVE THE NUMBER IN H.
         CALL    /THEN CALL THE OCTIN SUBROUTINE SO THAT
         OCTIN   /THE LO ADDRESS OF THE BREAKPOINT
         0       /CAN BE ENTERED.
         MOVLD   /SAVE THIS NUMBER IN L.
         CALL    /THEN SET THE BREAKPOINT AT THE
         SETBRK  /MEMORY LOCATION ADDRESSED BY REG-
         0       /ISTER PAIR H.
         MVIA    /NOW LOAD THE A REGISTER WITH THE
         303     /OP CODE FOR A "JMP" INSTRUCTION.
         STA     /STORE THIS OP CODE WHERE THE 8080
         050     /WILL BE VECTORED TO BY THE
         000     /"BREAKPOINT" INSTRUCTION.
         LXIH    /THEN LOAD REGISTER PAIR H WITH THE
         TRAP    /HI AND LO ADDRESS BYTES FOR THE
         0       /JUMP INSTRUCTION.
         SHLD    /AND STORE THIS 16-BIT ADDRESS IN
         051     /THE MEMORY LOCATIONS JUST AFTER
         000     /THE JMP OP CODE.
         JMP     /ONCE THE BREAKPOINT IS SET, JUMP
         CMDDEC  /BACK TO THE COMMAND DECODER OF
         0       /THE DEBUGGER.
SETBRK,  MOVAM   /GET THE "ORIGINAL" INSTRUCTION
         STA     /AND SAVE IT IN R/W MEMORY.
         ORGIN
         0
         MVIM    /THEN WRITE A RESTART INSTRUCTION OVER
         357     /THIS INSTRUCTION IN THE PROGRAM.
         SHLD    /SAVE THE ADDRESS WHERE THE BREAK-
         TEMPO   /POINT IS BEING SET IN
         0       /"TEMPO."
         RET     /THEN RETURN FROM SETBRK.
```

050, 000 051, and 000 052 (0028, 0029, and 002A) exist in R/W memory and are not used by the program being debugged.

After the breakpoint is set in Example 10-3, the 8080 returns from the SETBRK subroutine and loads the A register with the op code for the 8080 unconditional jump instruction, JMP. This is stored at the RST5 vector address, 000 050 (0028). The HI and LO address bytes for this instruction are then loaded into register pair H. These address bytes are stored in memory locations 000 051 and 000 052 (0029 and 002A). Note that when Example 10-3 is *assembled* (see Appendix B), a 16-bit address will be assigned to the symbolic address TRAP. This is the section of the debugger that must save all of the registers on the stack and print the contents of the registers. In another section of this chapter, you will see what the TRAP section of the debugger should look like.

Is it possible that we will never reach the breakpoint that is stored in the program? Yes, it is. Suppose that you insert a breakpoint at address 003 010 (0308) in Example 10-4. This means that the RST5 instruction will be written on top of the OUT instruction. Once this example is started at address 003 000 (0300), using the GO or EXECUTE command of your system monitor or debugger, the 8080 will execute the loop in the TTYI subroutine until one of the teletypewriter or crt keys is pressed. What will happen if the Z key is pressed? If the Z key is pressed, the microcomputer will input the ASCII value 132. This means that the 8080 will not return from the TTYI subroutine; instead, it will halt. However, the breakpoint is still "set" at memory location 003 010 (0308). To remove this breakpoint, we will assume that the debugger has a K command, which causes the breakpoint that was not reached to be removed. This means that the content of ORGIN (the original op code stored in the program) is written back into the program, using the content of TEMPO as the memory address.

Example 10-4: A Breakpoint Test Program

```
        *003 000
DEMO,   LXISP   /LOAD THE STACK POINTER WITH A R/W
        STACK   /MEMORY ADDRESS BECAUSE A SUBROUTINE
        0       /WILL BE CALLED.
        CALL    /GET A CHARACTER FROM THE
        TTYI    /TELETYPEWRITER.
        0
        ADI     /ADD 005 (05) TO THE ASCII VALUE
        005     /FOR THE KEY THAT IS PRESSED.
        OUT     /THEN OUTPUT THIS VALUE TO SOME
        300     /SEVEN-SEGMENT LED DISPLAYS.
        HLT     /HALT WHEN DONE.
TTYI,   IN      /INPUT THE UART'S STATUS.
        001
        ANI     /SAVE ONLY THE RECEIVER'S STATUS.
        001
        JZ      /IF A=001, A KEY IS PRESSED.
        TTYI    /IF A=000, NO KEY IS PRESSED.
        0       /IF NO KEY IS PRESSED, KEEP WAITING.
        IN      /A KEY IS PRESSED, SO INPUT THE
        000     /CHARACTER'S ASCII CODE.
        ANI     /SET THE PARITY BIT (D7) OF THE
        177     /ASCII VALUE TO 0.
        CPI     /WAS THE Z KEY OF THE TELETYPEWRITER
        132     /OR CRT PRESSED?
        RNZ     /NO, THEN RETURN WITH THE VALUE IN A.
        HLT     /HALT, THE Z KEY WAS PRESSED.
```

Clearing the Breakpoint

In Example 10-3, the 16-bit address of the breakpoint was saved in R/W memory at TEMPO. Therefore, when the K command of the debugger is entered, indicating that we want to "kill" or remove the breakpoint, we simply have to write the content of ORGIN, the op code previously stored, into the memory location addressed by the content of TEMPO. This causes the RST5 to be written over by the instruction op code that was originally in the program. The instructions listed in Example 10-5 remove the breakpoint from the program being debugged by doing just these operations.

Example 10-5: Removing a Breakpoint From a Program

```
CLEARB, LHLD    /LOAD REGISTER PAIR H WITH THE
        TEMPO   /ADDRESS WHERE THE BREAKPOINT
        0       /IS SET.
        LDA     /THEN LOAD THE A REGISTER WITH THE
        ORGIN   /OP CODE FOR THE INSTRUCTION THAT
        0       /WAS AT THIS ADDRESS.
        MOVMA   /SAVE THE OP CODE AT THE BREAKPOINT
        RET     /ADDRESS AND THEN RETURN.
```

Note that the K command is used to remove a breakpoint from a program (Table 10-1). This command can be used to remove the breakpoint if it is placed on the wrong instruction, or it can be used to remove the breakpoint after it is reached. Since CLEARB (Example 10-5) is a subroutine, the K command would cause it to be called, and then control would be returned to the command decoder within the debugger. You will soon see why we chose to make CLEARB a subroutine.

SAVING AND PRINTING THE CONTENTS OF THE REGISTERS

When the 8080 reaches the breakpoint and executes the RST5 instruction, the subroutine at 000 050 (0028) is called, which immediately "jumps" the 8080 to TRAP (Example 10-3). At TRAP, the registers are saved on the stack and the contents of the registers

Example 10-6: Saving the Registers When a Breakpoint Is Reached

```
/IF THE BREAKPOINT IS REACHED (THE RESTART INSTRUCTION
/EXECUTED), THE 8080 VECTORS TO 000 050, BECAUSE
/AN RST5 WAS USED FOR THE BREAKPOINT. AT 000 050,
/A JMP INSTRUCTION IS EXECUTED THAT CAUSES PRO-
/GRAM CONTROL TO BE TRANSFERRED TO "TRAP."

TRAP,   XTHL    /H AND L ON STACK, BREAKPOINT ADDRESS IN H AND L.
        DCXH    /DECREMENT THE ADDRESS.
        SHLD    /SAVE THE ADDRESS IN "BRKADD," BECAUSE
        BRKADD  /THIS IS THE ADDRESS WHERE THE
        0       /BREAKPOINT WAS REACHED.
        PUSHD   /THEN SAVE REGISTER PAIR D,
        PUSHB   /REGISTER PAIR B,
        PUSHPSW /AND A AND THE FLAGS ON THE STACK.
         •
         •
```

are also printed on the teletypewriter or crt. Example 10-6 contains the instructions that can be used to save the registers on the stack when the breakpoint is reached.

When the 8080 jumps to TRAP, the XTHL causes the content of register pair H to be put on the stack, and the return address on the stack (due to the RST5 instruction) is put in register pair H. If the breakpoint was reached at memory address 003 120 (0350), then register pair H now contains 003 121 (0351). Remember, the return address saved on the stack is one more than the address of the RST5 instruction. After the "return address" is exchanged into register pair H, it is decremented once and saved in BRKADD. The address where the breakpoint was *reached* is now stored in BRKADD. This address will be used in other sections of the debugger. Register pair H is already on the stack due to the XTHL instruction, so all of the remaining register pairs and the flags are saved on the stack. At this point, we might print out the state of the flags and the content of the A register, followed by the contents of the other general-purpose registers.

This means that at some point in TRAP we could pop the PSW off of the stack into register pair H. The eight-bit flag word would be popped into the L register, and the A register would be popped into the H register. The content of the L register could then be typed out on the teletypewriter or crt as a string of eight 1s and 0s, so that the state of all five flags can be observed. The content of the H register (the original content of the A register) could then be printed as either an octal or hexadecimal number, depending on the type of output subroutine that you have used. Likewise, the remaining registers could be popped off of the stack and printed. The instructions listed in Example 10-7 save the registers on the stack when the breakpoint is reached, and also print the state of the flags and the contents of the general-purpose registers on a teletypewriter or crt.

Example 10-7: Saving and Printing the Registers at TRAP

```
/IF THE BREAKPOINT IS REACHED (THE RESTART INSTRUCTION
/EXECUTED), THE 8080 VECTORS TO 000 050, BECAUSE
/A RST5 WAS USED FOR THE BREAKPOINT. AT 000 050,
/A JMP INSTRUCTION IS EXECUTED THAT CAUSES PRO-
/GRAM CONTROL TO BE TRANSFERRED TO "TRAP."

TRAP,   XTHL    /H AND L ON STACK, BREAKPOINT ADDRESS IN H AND L.
        DCXH    /DECREMENT THE ADDRESS.
        SHLD    /SAVE THE ADDRESS IN "BRKADD," BECAUSE
        BRKADD  /THIS IS THE ADDRESS WHERE THE
        0       /BREAKPOINT WAS REACHED.
        PUSHD   /THEN SAVE REGISTER PAIR D,
        PUSHB   /REGISTER PAIR B,
        PUSHPSW /AND A AND THE FLAGS ON THE STACK.
```

```
         POPH     /POP A AND THE FLAGS INTO H AND L.
         CALL     /THEN PRINT OUT THE EIGHT-BIT FLAG
         BIT      /WORD CONTAINED IN THE L REGISTER
         0        /AS A STRING OF ONES AND ZEROS.
         MOVAH    /THEN MOVE THE A REGISTER TO A
         CALL     /AND PRINT THE EQUIVALENT THREE-
         BINOCT   /DIGIT OCTAL NUMBER ON THE TELE-
         0        /TYPEWRITER OR CRT.
         MVID     /SET D TO 3, BECAUSE THERE ARE
         003      /3 OTHER REGISTER PAIRS TO PRINT.
REGOUT,  POPH     /POP A REGISTER PAIR INTO H AND L.
         MOVAH    /GET THE EIGHT MSB'S INTO A
         CALL     /AND PRINT THEIR OCTAL EQUIVALENTS
         BINOCT   /ON THE TELETYPEWRITER OR CRT.
         0
         MOVAL    /THEN LOAD A WITH THE EIGHT LSB'S
         CALL     /AND PRINT THEIR EQUIVALENT
         BINOCT   /OCTAL NUMBERS ON THE TELE-
         0        /TYPEWRITER OR CRT.
         DCRD     /DECREMENT THE REGISTER PAIR COUNT.
         JNZ      /IF THE COUNT IS NONZERO, THEN
         REGOUT   /JUMP BACK TO REGOUT SO THAT ANOTHER
         0        /REGISTER PAIR IS PRINTED OUT.
         JMP      /ALL OF THE REGISTERS AND THE FLAGS
         CMDDEC   /HAVE BEEN PRINTED OUT, SO GET
         0        /ANOTHER COMMAND.
BIT,     MVIC     /LOAD THE C REGISTER WITH THE NUMBER
         010      /OF "BITS" TO BE PRINTED.
NXTBIT,  MOVAL    /GET THE WORD TO BE "PRINTED."
         RLC      /ROTATE THE MSB INTO THE CARRY,
         MOVLA    /AND THEN SAVE THE WORD.
         MVIA     /LOAD THE A REGISTER WITH THE
         260      /VALUE FOR ASCII 0 (HEX B0).
         JNC      /IF THE CARRY IS ZERO, THEN PRINT
         ZERO     /A "0" ON THE TELETYPEWRITER OR
         0        /CRT. IF THE CARRY IS ONE,
         INRA     /PRINT A 1.
ZERO,    CALL     /PRINT THE CONTENT OF THE A REGISTER
         TTYOUT   /ON THE TELETYPEWRITER OR CRT.
         0
         DCRC     /THEN DECREMENT THE "BIT" COUNT.
         JNZ      /IF THE COUNT IS NONZERO, TEST
         NXTBIT   /ANOTHER BIT OF THE L REGISTER.
         0
         RET      /ALL EIGHT BITS HAVE BEEN PRINTED
                  /SO RETURN.
```

After the 8080 has saved all of the registers on the stack in Example 10-7, the PSW is popped off of the stack into register pair H. The L register contains the flag word when the BIT subroutine is called. At BIT, the C register is loaded with 010 (08), because there are eight bits within the flag word that have to be printed. The next three instructions have the net effect of rotating the content of the L register to the left, rotating the MSB of the L register into the carry. The A register is then loaded with the value for an ASCII 0. Remember, this instruction, MVIA, does not affect any of the 8080 flags. The state of the carry flag is then tested with the JNC instruction. Therefore, if the carry is a logic 0, the JNC to ZERO is executed and the 260 (B0) in the A register is transmitted to the teletypewriter and a 0 is printed. If the carry contains a 1, the JNC instruction is not executed, and the content of the A register is incremented by 1 to 261 (B1), the value for an ASCII 1. The 1 is then printed on the teletypewriter.

After a 1 or a 0 is printed, the bit count in the C register is decremented by 1. If the content of the C register is nonzero, the JNZ to NXTBIT is executed so that another bit within the L register can be rotated into the carry and tested. After all of the eight bits in the L register have been printed as 1s or 0s, the 8080 returns from the BIT subroutine. The content of the A register, which was popped off of the stack into the H register, is then printed on the teletypewriter using a binary-to-ASCII-based octal conversion subroutine.

There are still three other register pairs stored on the stack that must be printed. Therefore, after the content of the A register is

Example 10-8: A Typical Eight-Bit Binary-to-ASCII-Based, Octal Conversion Subroutine

```
/THIS SUBROUTINE CONVERTS THE BINARY VALUE STORED
/IN THE A REGISTER INTO A THREE-DIGIT, ASCII-BASED,
/OCTAL NUMBER.

BINOCT, MOVCA   /SAVE THE BINARY VALUE IN THE C REGISTER.
        ANI     /THE MOST-SIGNIFICANT DIGIT MUST BE
        300     /PRINTED FIRST.
        RLC     /ROTATE THE TWO MSB'S INTO THE
        RLC     /TWO LSB'S.
        CALL    /CALL THE BCDOUT SUBROUTINE, WHICH
        BCDOUT  /ADDS 260 (B0) TO THE CONTENT OF THE
        0       /A REGISTER AND PRINTS THE RESULT.
        MOVAC   /NOW THE MIDDLE DIGIT MUST BE PRINTED.
        ANI
        070
        RRC     /ROTATE BITS D5, D4, AND D3 INTO BITS
        RRC     /D2, D1, AND D0.
        RRC
        CALL    /ADD 260 (B0) TO THE CONTENT OF THE
        BCDOUT  /A REGISTER AND PRINT THE RESULT.
        0
        MOVAC   /NOW PRINT THE RIGHT-HAND AND
        ANI     /LEAST-SIGNIFICANT DIGIT.
        007
BCDOUT, ADI     /ADD 260 (B0) TO THE CONTENT
        260     /OF THE A REGISTER
        JMP     /AND PRINT THE RESULT.
        TTYOUT
        0
```

printed, the D register is loaded with the value 3, the number of register pairs that still must be printed. The register pairs are then popped off of the stack into register pair H (REGOUT), and the contents of registers H and L are printed in the form of two three-digit octal numbers. Each time the REGOUT loop is executed, the content of the D register is decremented by 1. Therefore, the REGOUT loop is executed three times and all the remaining registers are printed on the teletypewriter. A listing of a typical BINOCT subroutine is shown in Example 10-8. Example 10-8 is very similar to Example 6-8 in *8080/8085 Software Design–Book 1*[2].

The version of TRAP listed in Example 10-7 has one severe problem. Do you know what this is? By popping the registers off of the stack so that they can be printed, the values that were present in the registers when the breakpoint was reached are destroyed. This means that it will be impossible for the debugger to have a *continue* (execution) command. The debugger cannot load the registers with their original values before continuing on with the execution of the program being debugged. We will discuss a much better version of TRAP shortly and, using this version, we will be able to implement the continue command.

BREAKPOINT OPERATION

Suppose that we want to use a breakpoint to debug the program listed in Example 10-9. We want to place the breakpoint at the first OUT instruction. When the breakpoint has replaced this instruction, the program would appear like the one listed in Example 10-10.

Example 10-9: A Program To Test the Breakpoint

```
        *003 337
TEST,   MVIA    /LOAD THE A REGISTER WITH BCD
        231     /99 (OCTAL 231 = HEX 99).
        OUT     /OUTPUT THE VALUE TO TWO
        101     /SEVEN-SEGMENT LED DISPLAYS.
        ADI     /ADD ONE TO THIS NUMBER.
        001
        DAA     /DECIMAL ADJUST IT
        OUT     /AND OUTPUT THE RESULT TO TWO
        102     /OTHER SEVEN-SEGMENT LED DISPLAYS.
        HLT     /THEN STOP.
```

To replace the OUT with the breakpoint, we would have to specify the address of the OUT instruction op-code in a command such as 003 341 B. Once the breakpoint has been set, we can use the debugger to begin execution of the program; i.e., 003, 337 G or GO TO 03DF. When this command is entered, the 8080 jumps to 003 337 (03DF) and executes the instruction stored in this memory location. This means that the MVIA instruction is executed, so the A register is loaded with the immediate data byte, 231 (99). The RST5 instruction, the breakpoint instruction, is then executed. The 8080 calls the "subroutine" at 000 050 (0028). The JMP instruction in this memory location causes program control to be transferred to TRAP.

Example 10-10: The Program With the Breakpoint in Place

```
        *003 337
TEST,   MVIA    /LOAD THE A REGISTER WITH BCD
        231     /99 (OCTAL 231 = HEX 99).
        RST5    /*****BREAKPOINT INSTRUCTION*****
        101     /SEVEN-SEGMENT LED DISPLAYS.
        ADI     /ADD ONE TO THIS NUMBER.
        001
        DAA     /DECIMAL ADJUST IT
        OUT     /AND OUTPUT THE RESULT TO TWO
        102     /OTHER SEVEN-SEGMENT LED DISPLAYS.
        HLT     /THEN STOP.
```

At TRAP, register pair H is exchanged onto the stack, and the return address due to the RST5 instruction is exchanged into register pair H. This return address is decremented by 1 and is saved in BRKADD. This is the address where the breakpoint was reached. Register pairs D and B, and register A and the flags are then saved on the stack. The 8080 then prints the state of the flags and the contents of the registers on the teletypewriter or crt. After all of the flags and registers have been printed, the 8080 jumps back to the command decoder contained in the debugger. The following is a sample printout obtained by using the debugger listed in *DBUG: An 8080 Interpretive Debugger*[1]:

```
003 341 B
003 337 G
003 341
SZ1 P 2  A   B   C   D   E   H   L   M   S P     C S
01000110 231 000 000 000 000 000 000 024 177 372 107 125
```

Has the breakpoint been cleared from the sample program by the TRAP section of the debugger (Example 10-7)? Has the breakpoint instruction (RST5) been replaced by the original OUT instruction? No, it has not. However, to replace the RST5 with the OUT instruction, we could enter the K command so that the breakpoint is removed (Example 10-5). At some time while using the breakpoint to debug a program, we may forget to remove the breakpoint. It would be far easier to rewrite the TRAP section of the debugger so that the breakpoint is removed by calling the CLEARB subroutine before the 8080 jumps back to the command decoder (CMDDEC). This is done just after the REGOUT loop in Example 10-11. In Example 10-11, we have not shown the entire TRAP section of the debugger, only the last section of the instruction sequence that prints out the contents of the registers, the instructions that clear the breakpoint, and finally the jump back to the command decoder.

Example 10-11: Clearing the Breakpoint After It Is Reached

```
          •
          •
          MOVAL   /THEN LOAD A WITH THE EIGHT LSB'S
          CALL    /AND PRINT THEIR EQUIVALENT
          BINOCT  /OCTAL NUMBERS ON THE TELE-
          0       /TYPEWRITER OR CRT.
          DCRD    /DECREMENT THE REGISTER PAIR COUNT.
          JNZ     /IF THE COUNT IS NONZERO, THEN
          REGOUT  /JUMP BACK TO REGOUT SO THAT ANOTHER
          0       /REGISTER PAIR IS PRINTED OUT.
          CALL    /NOW CLEAR THE BREAKPOINT BY WRITING
          CLEARB  /THE ORIGINAL INSTRUCTION BACK IN
          0       /THE PROGRAM BEING DEBUGGED.
          JMP     /ALL OF THE REGISTERS AND THE FLAGS
          CMDDEC  /HAVE BEEN PRINTED OUT, SO GET
          0       /ANOTHER COMMAND.
CLEARB,   LHLD    /LOAD REGISTER PAIR H WITH THE
          TEMPO   /ADDRESS WHERE THE BREAKPOINT
          0       /IS SET.
          LDA     /THEN LOAD THE A REGISTER WITH THE
          ORGIN   /OP CODE FOR THE INSTRUCTION THAT
          0       /WAS AT THIS ADDRESS.
          MOVMA   /SAVE THE OP CODE AT THE BREAKPOINT
          RET     /ADDRESS AND THEN RETURN.
```

NONDESTRUCTIVELY PRINTING THE CONTENTS OF THE REGISTERS

As we mentioned previously, the TRAP section of the debugger destroys the contents of the registers when they are printed out. This occurs because the registers are popped off of the stack. Using Example 10-9 again, let us assume that a breakpoint was placed on the first OUT instruction, and that the 8080 began program execution at 003 337. When the breakpoint is reached, the contents of the registers are printed out and the breakpoint is removed (the OUT instruction is written back into the program). The 8080 then returns to the command decoder contained in the debugger. Now we want to place the breakpoint on the second OUT instruction and have the 8080 *continue* executing this program, starting at the *first OUT instruction.* This means that we want to *continue* program execution from the point where the last breakpoint was set. By going through these steps, we will be able to observe the effect of the OUT 101 instruction.

To do this now would require restarting the program at the beginning (003 337, 03DF). In our test program this is not difficult, but in some programs it may be very difficult, since the programs may be long and they may depend on real-time events.

Since the address of the last location of the breakpoint is contained in TEMPO, two R/W memory locations, it would not be difficult for the microcomputer to determine where it needs to "restart" execution of the program being debugged. However, to continue program execution, we also have to get all of the flags and registers back the way they were when the first breakpoint was reached. The problem is that the TRAP section of the debugger (Example 10-7) did not save the contents of the registers so that the 8080 could continue program execution, since the registers were popped off of the stack so that they could be printed on the teletypewriter. What we must be able to do is to print the contents of the registers while still saving them on the stack. To do this, we *do not* pop a 16-bit word off of the stack, print the two register values, and then push the same 16-bit word back on the stack. If this were done, it would be very difficult to get any of the other 16-bit words off of the stack. Of course, the reason that the registers have to be saved on the stack in the first place is because the conversion and output subroutines have to use some registers. Since the "stack" is really a section of R/W memory, there are other methods that can be used to access the register values after they are pushed onto the stack.

The instructions listed in Example 10-12 save the 8080 registers on the stack, and then load register pair H with the R/W memory address of the stack pointer. Now, by using MOV-type instructions, the 8080 can read a value from memory into one of the general-purpose registers, and then call the BIT, BINOCT, or BINHEX subroutines, depending on the form in which the number must be printed on the teletypewriter or crt. Since the MOV-type instructions do not affect the stack, it is very easy to print out the contents of the registers and the state of the flags, while leaving the register information on the stack. Let us examine the stack area and see what the actual storage order of the registers is. This will enable us to write a sequence of instructions that can access these values and print them on the teletypewriter or crt. We will assume that register pair H is the first register pair that is pushed onto the stack, and that the PSW is the last "register pair" that is pushed onto the stack.

In Table 10-2, we have assumed that the SP pointed to memory address 024 100 (1440) before any of the PUSH-type instructions at the TRAP section of the debugger were executed. If the instructions listed in Example 10-12 are now executed, what memory address will be contained in register pair H when the DADSP instruction is executed? Register pair H will contain the address 024 070

Example 10-12: Using Register Pair H To Access the Registers

```
/NOW THE "TRAP" SECTION OF THE DEBUG PROGRAM
/ADDS THE SP TO REGISTER PAIR H. MEMORY-REFERENCE IN-
/STRUCTIONS CAN THEN BE USED TO ACCESS THE REGISTER
/VALUES ON THE "STACK" WITHOUT DISTURBING THE STACK.

TRAP,   XTHL     /H AND L ON STACK, BREAKPOINT ADDRESS IN H AND L.
        DCXH     /DECREMENT THE ADDRESS.
        SHLD     /SAVE THE ADDRESS IN "BRKADD," BECAUSE
        BRKADD   /THIS IS THE ADDRESS WHERE THE
        0        /BREAKPOINT WAS REACHED.
        PUSHD    /THEN SAVE REGISTER PAIR D,
        PUSHB    /REGISTER PAIR B,
        PUSHPSW  /AND A AND THE FLAGS ON THE STACK.
        LXIH     /NOW LOAD REGISTER PAIR H WITH
        000      /000 000 (0000).
        000
        DADSP    /ADD THE SP TO REGISTER PAIR H.
          •
          •
```

(1438), the memory location used to store the flag word of the PSW. As you can see from Table 10-2, we cannot simply read a value from the memory location addressed by register pair H, print the value on the teletypewriter, and increment the memory address. This cannot be done because we want to print the flag word, then registers A, B, C, D, E, H, and L. This is *not* the order in which the registers have been written into memory by the PUSH instructions. The sequence of instructions that accesses these eight-bit values and prints them in the desired order is listed in Example 10-13.

Table 10-2. The Order of the Registers on the Stack

	Octal Stack Address		Hexadecimal Stack Address
Initial SP	024 100		1440
	024 077	Register H	143F
	024 076	Register L	143E
	024 075	Register D	143D
	024 074	Register E	143C
	024 073	Register B	143B
	024 072	Register C	143A
	024 071	Register A	1439
Current SP	024 070	The Flag Word	1438

The instructions in Example 10-13 may be somewhat difficult to follow, but they do cause the 8080 to print the registers on the teletypewriter in the desired order. The important point to remember is that since MOV-type instructions are used to *read* the register values from memory, the register values are still stored in memory after they are printed on the teletypewriter or crt. Likewise, the stack-pointer value has not been altered either.

Example 10-13: Printing the Contents of the Registers in the Desired Order When the Breakpoint Is Reached

```
/NOW THE "TRAP" SECTION OF THE DEBUG PROGRAM
/ADDS THE SP TO REGISTER H. MEMORY-REFERENCE IN-
/STRUCTIONS CAN THEN BE USED TO ACCESS THE REGISTER
/VALUES ON THE "STACK" WITHOUT DISTURBING THE STACK.

TRAP,    XTHL     /H AND L ON STACK, BREAKPOINT ADDRESS IN H AND L.
         DCXH     /DECREMENT THE ADDRESS.
         SHLD     /SAVE THE ADDRESS IN "BRKADD," BECAUSE
         BRKADD   /THIS IS THE ADDRESS WHERE THE
         0        /BREAKPOINT WAS REACHED.
         PUSHD    /THEN SAVE REGISTER PAIR D,
         PUSHB    /REGISTER PAIR B,
         PUSHPSW  /AND A AND THE FLAGS ON THE STACK.
         LXIH     /NOW LOAD REGISTER PAIR H WITH
         000      /000 000 (0000).
         000
         DADSP    /ADD THE SP TO REGISTER PAIR H.
         MOVAM    /NOW READ THE FLAG WORD FROM MEMORY
         CALL     /AND PRINT THE FLAGS AS ONES AND ZEROS.
         BIT      /THIS IS A NEW VERSION OF THE
         0        /"BIT" SUBROUTINE SEEN PREVIOUSLY.
         INXH     /INCREMENT THE ADDRESS IN REGISTER PAIR H.
         MOVAM    /GET THE "A REGISTER"
         CALL     /AND PRINT IT OUT IN THE FORM OF
         BINOCT   /A THREE-DIGIT OCTAL NUMBER.
         0
         MVID     /LOAD THE D REGISTER WITH 003
         003      /(THREE REGISTER PAIRS TO BE PRINTED).
REGPR,   MVIE     /LOAD THE E REGISTER WITH THE NUMBER
         002      /OF REGISTERS IN A REGISTER PAIR.
         INXH     /INCREMENT THE MEMORY ADDRESS
         INXH     /TWICE.
NXTREG,  MOVAM    /GET AN EIGHT-BIT WORD FROM MEMORY
         CALL     /AND PRINT ITS OCTAL EQUIVALENT
         BINOCT   /ON THE TELETYPEWRITER OR CRT.
         0
         DCXH     /DECREMENT THE MEMORY ADDRESS
         DCRE     /AND THE REGISTER COUNT.
         JNZ      /IF THE COUNT IS NONZERO, PRINT
         NXTREG   /THE CONTENT OF THE OTHER REGISTER
         0        /IN THE REGISTER PAIR.
         INXH     /PRINTED ONE COMPLETE REGISTER PAIR
         INXH     /SO INCREMENT THE MEMORY ADDRESS TWICE.
         DCRD     /THEN DECREMENT THE REGISTER PAIR COUNT.
         JNZ      /IF THE COUNT IS NONZERO, THEN
         REGPR    /THE CONTENT OF ANOTHER REGISTER PAIR
         0        /MUST BE PRINTED.
         CALL     /ALL THE PRINTING HAS BEEN COMPLETED
         CLEARB   /SO REMOVE THE BREAKPOINT (THE RST5)
         0        /FROM THE PROGRAM.
```

```
        JMP     /THEN GET ANOTHER COMMAND.
        CMDDEC
        0
BIT,    MOVDA   /SAVE THE EIGHT-BIT NUMBER IN D.
        MVIC    /LOAD THE C REGISTER WITH THE NUMBER
        010     /OF "BITS" TO BE PRINTED.
NXTBIT, MOVAD   /GET THE WORD TO BE "PRINTED."
        RLC     /ROTATE THE MSB INTO THE CARRY.
        MOVDA   /THEN SAVE THE RESULT IN REGISTER D.
        MVIA    /LOAD THE A REGISTER WITH THE VALUE
        260     /FOR AN ASCII 0 (HEX B0).
        JNC     /IF THE CARRY IS ZERO, PRINT A
        ZERO    /ZERO ON THE TELETYPEWRITER OR CRT.
        0       /IF THE CARRY IS A LOGIC ONE,
        INRA    /THEN INCREMENT THE 260 TO 261 (B0 TO B1).
ZERO,   CALL    /PRINT THE CONTENT OF THE A REGISTER
        TTYOUT  /ON THE TELETYPEWRITER OR CRT.
        0
        DCRC    /DECREMENT THE "BIT" COUNT.
        JNZ     /IF THE COUNT IS NONZERO, TEST
        NXTBIT  /ANOTHER BIT IN THE D REGISTER.
        0       /RETURN WHEN ALL EIGHT BITS IN THE
        RET     /D REGISTER HAVE BEEN PRINTED.
```

One of the characteristics of the POP-type instructions that you should remember is that the values are nondestructively read from memory. This means that the values could be popped off of the stack and printed. The values are still stored in R/W memory. However, once the contents of the registers have been printed, if *any* subroutines are called, the return addresses will be saved on the stack, destroying the contents of the registers. Therefore, it is easier to add the stack pointer to register pair H, which has been initialized to 0, and then use MOV-type instructions to read the values from memory. If any subroutines are called after this is done, the return addresses will be saved on the stack *below* the contents of the registers and the flag word.

ADDING A CONTINUE COMMAND TO THE DEBUGGER

With all of the registers still saved on the stack after they are printed on the teletypewriter, the continue command simply has to restore the registers to their initial values, and then jump to the breakpoint address to restart or continue executing the program being debugged. This address is stored in BRKADD (see Example 10-3).

When the *continue* command is entered into the microcomputer and detected by the command decoder of the debugger, the 8080 jumps to CONTIN in Example 10-14. At CONTIN, the PSW and register pairs B and D are popped off of the stack (in the proper order, of course). These registers were pushed onto the stack in the order established by the instruction sequence in Example 10-13. The original 16-bit value that was in register pair H when the breakpoint was reached is still on the stack. Register pair H is then loaded with the breakpoint address, which is the address of the next instruction that must be executed in the program being debugged. In Examples 10-9 and 10-10, this was the first OUT instruction. *Remember, this instruction was not executed when the breakpoint was reached. Also, the TRAP section of the debugger removes the breakpoint after the contents of the registers are printed out.* Once the breakpoint address is loaded into register pair H, the XTHL puts this address on the stack *and* restores register pair H to its original value when the breakpoint was reached. The RET instruction then pops BRKADD off of the stack and into the program counter (PC). The 8080 then continues to execute the program being debugged, with *all* of the registers restored to their original values.

Using the BREAK (Example 10-3), TRAP (Example 10-13), and CONTIN (Example 10-14) instructions, would it be possible for us to set a new breakpoint one, two, or 10 instructions away from the previous breakpoint, and then use the continue command so that we could observe the effect the one, two, or 10 instructions had on the contents of the registers? Yes, this is possible. Let us examine the operations (commands) that you would have to perform. After setting the breakpoint at some instruction, the debugger would be used to transfer program control to the beginning of the program being debugged. When the breakpoint is reached, the contents of the registers are printed out. You would then set the breakpoint a few instructions away from the last breakpoint, using the BREAK instructions. *These instructions cause the address where the breakpoint is set to be saved in TEMPO.* The continue command would then be entered into the debugger so that the CONTIN instructions are executed. These instructions pop all of the register values off of the stack and into the proper registers, *and then transfer program control to the instruction that is stored in memory at the address where the last breakpoint was reached.* Therefore, BRKADD contains the address where program execution should continue from, and TEMPO is used to store the address where the breakpoint is set. It is extremely important to realize the distinction between these two addresses.

When the instructions at TRAP (Example 10-13) are executed to save the registers, which stack are the registers saved on? This is a difficult question. No doubt, the debugger executes an LXISP instruction when it is first started. However, the program being debugged probably has an LXISP instruction at its beginning also. Therefore, if the program to be debugged is started from its beginning, using the debugger GO command, the SP will be loaded last by the LXISP instruction in the program being debugged. Therefore, at TRAP the registers will be saved on the stack "belonging to" the program that is being debugged. This occurs simply because the program being debugged executed the last LXISP instruction.

Example 10-14: The Instructions That Cause the 8080 To "Continue"

```
/THE CONTINUE COMMAND CAUSES THE 8080 TO CONTINUE
/EXECUTING THE USER'S PROGRAM, STARTING AT THE
/ADDRESS OF THE LAST BREAKPOINT.

CONTIN, POPPSW  /POP THE A REGISTER AND FLAGS,
        POPB    /REGISTER PAIR B,
        POPD    /AND REGISTER PAIR D OFF OF THE STACK.
        LHLD    /LOAD REGISTER PAIR H WITH THE
        BRKADD  /ADDRESS WHERE THE LAST BREAKPOINT
        0       /WAS PLACED.
        XTHL    /PUT IT ON THE STACK, REGISTER PAIR
        RET     /H THE WAY IT WAS, THEN JUMP TO THE
                /BREAKPOINT ADDRESS.
```

This can lead to problems. After the registers are saved on the stack and are printed, the 8080 returns to the command decoder within the debugger so that another command can be entered and executed. This means that if any call instructions are executed, the return addresses are saved *on the stack that is established and used by the user's program.* When return instructions are executed, these return addresses are popped off of the user's stack. However, if the debugger does not clean its data values and return addresses off of the stack before the continue command is entered, these values will be popped off of the stack at CONTIN and will be used as the "original" register values. Of course, a program should maintain a "clean" stack, but finding all of the "bugs" in a debugger can be difficult. How do you debug a debugger?

Another problem with the debugger using the stack established by the user, and a more important problem, is that the user may not have allocated enough room on the stack for both the program data values and return addresses *and* the debugger data values and return addresses. Suppose the debugger needs 15 levels on the stack (30 R/W memory locations) to operate. If the user has only allocated 10 levels for the stack (20 R/W memory locations), as determined by the second and third bytes of the LXISP instruction in the user's program, then the 15 levels of stack that the debugger requires may write over some of the user's program or data values. This may cause the user's program to become "lost." The only solution to this is to use two *independent* stacks, one for the user's program and one for the debugger. However, only one stack will be in use at any time.

Having two independent stacks increases the complexity of the debugger breakpoint software (TRAP and CONTIN), but that is the price that must be paid. The debugger stack will be used when the debugger is being executed, and the user's stack will be used when the user's program is being executed by the 8080. To do this, stack switching instructions have to be executed at TRAP so that the computer switches from the user's stack to the debugger stack, and at CONTIN so that the 8080 switches from the debugger stack back to the user's stack. The new TRAP instructions are listed in Example 10-15.

Example 10-15: The New TRAP With Stack-Switching Instructions

```
/THE TRAP SECTION OF THE DEBUGGER NOW SAVES ALL OF THE REG-
/ISTERS ON THE USER'S STACK. THE STACK POINTER IS ALSO
/SAVED IN R/W MEMORY. A NEW STACK IS THEN SET UP FOR THE
/EXCLUSIVE USE OF THE DEBUGGER.

TRAP,   XTHL    /H AND L ON STACK, BREAKPOINT ADDRESS IN H AND L.
        DCXH    /DECREMENT THE ADDRESS.
        SHLD    /SAVE THE ADDRESS IN "BRKADD," BECAUSE
        BRKADD  /THIS IS THE ADDRESS WHERE THE
        0       /BREAKPOINT WAS REACHED.
        PUSHD   /THEN SAVE REGISTER PAIR D,
        PUSHB   /REGISTER PAIR B,
        PUSHPSW /AND A AND THE FLAGS ON THE STACK.
        LXIH    /NOW LOAD REGISTER PAIR H WITH
        000     /000 000 (0000).
        000
        DADSP   /ADD THE SP TO REGISTER PAIR H.
        SHLD    /SAVE THE "USER'S SP" IN R/W MEMORY
        USERSP  /FOR LATER USE. REGISTER PAIR H
        0       /STILL CONTAINS THE SP ADDRESS.
        LXISP   /THEN SET A NEW STACK POINTER FOR THE
        STACK   /DEBUGGER. NONE OF THE USER'S REGISTERS CAN
        0       /NOW BE "CLOBBERED" BY A RUNAWAY STACK.
        MOVAM   /NOW READ THE FLAG WORD FROM MEMORY
        CALL    /AND PRINT THE FLAGS AS ONES AND ZEROS.
        BIT     /THIS IS A NEW VERSION OF THE
        0       /"BIT" SUBROUTINE SEEN PREVIOUSLY.
         •
         •
```

Only two instructions have to be added to TRAP. After all of the registers are saved on the *user's stack* (four levels, eight R/W memory locations), the user's stack pointer is added to register pair H and is saved in two R/W memory locations by an SHLD instruction. After the SHLD instruction has been executed, register pair H will still contain the contents of the SP, so the contents of the registers can still be read from R/W memory, using the address in register pair H. The debugger then executes an LXISP instruction, so that a new stack is established for the *exclusive use* of the debugger. Any stack-related operations that the debugger now performs (i.e., calls, returns, pushes, or pops) will use this stack and not the user's stack. Note that the debugger still used the user's stack to save the registers.

The CONTIN section of the debugger must also be changed so that the debugger switches back to the user's stack *before* popping the contents of the registers off of this stack. The new CONTIN instructions are listed in Example 10-16. In this example, the user's SP is read from USERSP, two R/W memory locations, into register pair H, and the SP is then loaded with this value when the SPHL instruction is executed. The PSW and register pairs B and D are then popped off of the stack and into their respective destinations. Register pair H is then loaded with the content of BRKADD, which is the address where the breakpoint was reached and where the 8080 must continue executing the user's program. The XTHL instruction then exchanges the original content of register pair H, which is on the stack, with the last breakpoint address. The RET instruction loads the program counter with this address.

Example 10-16: CONTIN With Stack-Switching Instructions

```
/NOW THAT THE USER'S SP HAS BEEN SAVED IN R/W MEMORY,
/THE "CONTINUE" COMMAND MUST LOAD THE STACK POINTER
/WITH THIS ADDRESS BEFORE RETRIEVING THE REGISTERS
/FROM THE STACK AND CONTINUING PROGRAM EXECUTION.

CONTIN, LHLD    /LOAD REGISTER PAIR H WITH THE
        USERSP  /USER'S SP.
        0
        SPHL    /THEN LOAD THE SP WITH THIS VALUE.
        POPPSW  /POP A AND THE FLAGS OFF OF THE STACK.
        POPB    /AND THEN POP REGISTER PAIR B AND
        POPD    /REGISTER PAIR D.
        LHLD    /LOAD REGISTER PAIR H WITH THE
        BRKADD  /BREAKPOINT ADDRESS (WHERE WE
        0       /CONTINUE FROM). EXCHANGE WITH
        XTHL    /REGISTER PAIR H ON THE STACK.
        RET     /POP "BRKADD" INTO THE PROGRAM COUNTER.
```

The modifications to TRAP cause only four levels on the user's stack to be used to save the contents of the various registers. However, in some cases even this portion of the user's stack cannot be used, simply due to a lack of available R/W memory locations. This means that the contents of the registers must be saved on a third stack, or the debugger stack. If this is done, TRAP and CONTIN (Examples 10-15 and 10-16) will have to be modified. The simplest solution is for the user to allocate more memory for the stack. If this cannot be done, then the register values can be saved on the debugger stack.

SINGLE-STEPPING—EXECUTING ONE INSTRUCTION AT A TIME

If we have the program listed in Example 10-17 stored in R/W memory, it would be convenient if we could *easily* execute one instruction at a time and see the effect the single instruction had on the 8080 general-purpose registers. This means that we would place a breakpoint at a particular instruction, execute the program, and note the contents of the various registers. The breakpoint would then have to be advanced one instruction in the program, and the continue command entered.

Example 10-17: A Sample Program To Be Debugged

```
/THIS PROGRAM IS USED AS THE EXAMPLE FOR EXECUTING
/ONE INSTRUCTION AT A TIME (SINGLE-STEPPING).

        *003 000
START,  LXISP   /LOAD THE STACK POINTER
        300     /WITH A R/W MEMORY ADDRESS.
        003
        LXIH    /THEN LOAD REGISTER PAIR H WITH A
        030     /MEMORY ADDRESS (043 030 = 2318).
        043
        MOVAM   /GET AN EIGHT-BIT VALUE FROM MEMORY.
        ADDB    /ADD REGISTER B TO THIS VALUE.
        OUT     /OUTPUT THE RESULT TO OUTPUT
        203     /PORT 203 (83).
        HLT     /THEN HALT PROGRAM EXECUTION.
```

To execute one instruction at a time in Example 10-17, we would first place the breakpoint at the LXIH instruction at memory location 003 003 (0303) and begin executing the program at 003 000 (0300). When the breakpoint is reached, the user's stack will have been set by the LXISP instruction. The contents of all the registers are then printed out. The breakpoint will then have to be placed at the MOVAM instruction at 003 006 (0306), and the continue command entered. The effect that the LXIH has on the registers can then be observed when the registers are printed out. To step through the remainder of the program, the breakpoints would have to be placed sequentially in the memory locations used to store the op codes for the MOVAM, ADDB, OUT, and HLT instructions. By setting the breakpoint at each instruction and using the continue command of the debugger, we *single-step* through the entire program (execute one instruction at a time and note the effect that the instruction has on the registers).

One limitation of this process is that we have to determine the address of the memory location used to store the op code of each instruction. If you are unfamiliar with the instruction op codes of the 8080, or you are debugging someone else's program, this process of single-stepping can be long and tedious. A much more sophisticated and complex technique would be to program the 8080-based debugger to *calculate* the memory address at which the next instruction to be executed is stored. This means that instructions within the debugger must be able to determine whether an instruction is one, two, or three bytes. This means that the debugger must "know" that an MVIA instruction is a two-byte instruction, and that a JMP instruction is a three-byte instruction.

In Table 10-3, we have listed all of the one-, two-, and three-byte instructions of the 8080 along with their octal op codes. Note that some of the instructions are listed as a *class* of instructions; for instance, all of the MOV-type instructions are one-byte instructions, so they are simply listed as MOV*DS* (*D* = *destination register*, S = *source register*).

Once the "length" of each instruction is known, it is easy for the 8080 to calculate the address to which the next breakpoint must be moved. If X is the address of the last breakpoint and we need to single-step through a two-byte instruction, then the next breakpoint must be placed at address X + 2. For a three-byte instruction, the breakpoint address would be address X + 3. This ignores the possibility that a jump, call, restart, return, or PCHL instruction is executed. For now, to keep the debugger as simple as possible, we will ignore those instructions that have the capability of transferring program control (program execution).

If we want to single-step through Example 10-17, we would place the breakpoint at 003 003 (0303), where the LXIH instruction op code is stored, and then begin program execution at 003 000 (0300) using the G or GO command of the debugger. The LXISP instruction is then executed and the breakpoint reached. Since the LXISP instruction does not affect any of the 8080 general-purpose registers, we cannot predict what will be in these registers when the breakpoint is reached. When the *STEP* command is entered into the debugger, the breakpoint has to be set at address 003 006 (0306), and then the 8080 has to execute the instruction stored in memory at the previous breakpoint address of 003 003 (0303). This means that when the 8080 is to single-step through the instruction, it has to read the instruction op code at the previous breakpoint address and, based on this op code, determine if the new breakpoint must be set one, two, or three memory locations away. Once the new breakpoint is set at this new address, the "continue" sequence of instructions is executed, so that all of the

Table 10-3. The One-, Two-, and Three-Byte Instructions of the 8080

Mnemonic	Octal Op Code		Mnemonic	Octal Op Code
One-Byte Instructions				
MOV**DS**	1**DS**	**D = S = 0,1,2, 3,4,5,6,7**	RLC	007
			RRC	017
ADD**S**	20**S**		RAL	027
ADC**S**	21**S**		RAR	037
SUB**S**	22**S**		HLT	166
SBB**S**	23**S**		NOP	000
ANA**S**	24**S**		EI	373
XRA**S**	25**S**		DI	363
ORA**S**	26**S**			
CMP**S**	27**S**		STC	067
			CMC	077
INR**S**	0**S**4		DAA	047
DCR**S**	0**S**5		CMA	057
PUSH**RP**	3**R**5	**R = 0,2,4,6**	XCHG	353
POP**RP**	3**R**1	**R = 0,2,4,6**	XTHL	343
INX**RP**	0**R**3	**R = 0,2,4,6**	PCHL	351
DCX**RP**	0**R**3	**R = 1,3,5,7**	SPHL	371
STAX**RP**	0**R**2	**R = 0,2**		
LDAX**RP**	0**R**2	**R = 1,3**	RET	311
DAD**RP**	0**R**1	**R = 1,3,5,7**	Conditional Returns 3**X**0 **X = 0,1,2,3,4,5,6,7**	
RST**n**	3**N**7	**N = 0,1,2,3, 4,5,6,7**		
Two-Byte Instructions				
IN	333		ADI	306
OUT	323		ACI	316
			SUI	326
MVI**D**	0**D**6	**D = 0,1,2,3, 4,5,6,7**	SBI	336
			ANI	346
			XRI	356
			ORI	366
			CPI	376
Three-Byte Instructions				
JMP	303		LXI	0**R**1
Conditional Jumps	3**X**2			
CALL	315		STA	062
Conditional Call	3**X**4		LDA	072
	X = 0,1,2,3, 4,5,6,7		SHLD	042
			LHLD	052
			R = 0,2,4,6	

registers are restored to their original values *before the single instruction is executed.*

To keep the debugger simple, we will not let it single-step through jump, call, return, restart, or PCHL instructions. By looking at the octal op codes of the 8080 instructions, it is very easy to see how the 8080 can determine the number of bytes for each instruction.

The STEP instructions in Example 10-18 determine the number of bytes for the instruction to be executed, set a breakpoint immediately after this instruction, restore the user's stack and the register values, and then jump to the instruction in the program being debugged. After executing this instruction, the 8080 is "vectored" to TRAP by the RST5 instruction. After the first breakpoint is set, and the GO command of the debugger is entered into the microcomputer, we can begin to single-step the debugger through one instruction. To do this, we press the S key (for the STEP command) on the teletypewriter or crt. The command decoder within the debugger then transfers control to STEP (Example 10-18), where the 8080 determines the number of bytes for the instruction op code stored at the previous breakpoint address. If the debugger cannot single-step through the particular instruction, it will print a question mark on the teletypewriter or crt and then return to the command decoder within the debugger. If this happens, you will have to manually move the breakpoint around the transfer-of-control instruction. Note that the instructions in STEP comprise an *instruction decoder*, which is similar in function to a simple command decoder.

Example 10-18: Determining the Number of Bytes for Each 8080 Instruction

```
/THIS SECTION OF THE DEBUGGER DETERMINES THE NUMBER
/OF BYTES FOR EVERY 8080 INSTRUCTION. HOWEVER, SOME IN-
/STRUCTIONS CANNOT BE EXECUTED USING THE SINGLE-STEP
/FEATURE, INCLUDING JUMP, CALL, RETURN, RESTART, AND PCHL
/INSTRUCTIONS (ANY INSTRUCTION THAT TRANSFERS PROGRAM
/CONTROL).

STEP,   LDA     /LOAD THE A REGISTER WITH THE OP CODE
        ORGIN   /FOR THE INSTRUCTION THAT WAS STORED
        0       /IN MEMORY AT THE BREAKPOINT ADDRESS.
        MOVBA   /SAVE THE OP CODE IN B ALSO.
        LHLD    /THEN LOAD REGISTER PAIR H WITH THE
        BRKADD  /ADDRESS WHERE THE BREAKPOINT WAS
        0       /REACHED.
        CPI     /IS THE INSTRUCTION AN OUT?
        323
        JZ      /YES, THEN IT IS A TWO-BYTE
        TWOBYT  /INSTRUCTION.
        0
        CPI     /IS THE INSTRUCTION AN IN?
        333
        JZ      /YES, THEN IT IS A TWO-BYTE
        TWOBYT  /INSTRUCTION.
        0
        CPI     /IS THE INSTRUCTION A JMP?
        303
        JZ      /YES, THEN WE CANNOT SINGLE STEP
        CANTDO  /THROUGH IT. PRINT A ? ON THE
        0       /TELETYPEWRITER OR CRT.
        CPI     /IS THE INSTRUCTION A CALL?
        315
        JZ      /YES, THEN WE CANNOT SINGLE-STEP
        CANTDO  /THROUGH IT. PRINT A ? ON THE
        0       /TELETYPEWRITER OR CRT.
        CPI     /IS THE INSTRUCTION A RET?
        311
        JZ      /YES, THEN WE CANNOT SINGLE-STEP
        CANTDO  /THROUGH IT. PRINT A ? ON THE
        0       /TELETYPEWRITER OR CRT.
        CPI     /IS THE INSTRUCTION A PCHL?
        351
        JZ      /YES, THEN WE CANNOT SINGLE-STEP
        CANTDO  /THROUGH IT. PRINT A ? ON THE
        0       /TELETYPEWRITER OR CRT.
        ANI     /SAVE ONLY BITS D7, D6, D2, D1,
        307     /AND D0 IN THE A REGISTER.
        CPI     /IS THE INSTRUCTION ONE OF THE RESTARTS?
        307
        JZ      /YES, THEN WE CANNOT SINGLE-STEP
        CANTDO  /THROUGH IT. PRINT A ? ON THE
        0       /TELETYPEWRITER OR CRT.
        CPI     /IS IT AN IMMEDIATE MATH OR LOGICAL
        306     /INSTRUCTION?
        JZ      /YES, THEN IT IS A TWO-BYTE
        TWOBYT  /INSTRUCTION.
        0
        CPI     /IS IT AN IMMEDIATE MOVE INSTRUCTION?
        006
        JZ      /YES, THEN IT IS A TWO-BYTE
        TWOBYT  /INSTRUCTION.
        0
        CPI     /IS IT AN STA, LDA, SHLD, OR LHLD
        002     /(DIRECT LOAD) INSTRUCTION?
        JZ      /MAYBE IT IS, MAKE ANOTHER TEST.
        DLOAD
        0
        CPI     /IS IT AN LXIB, LXID, LXIH, OR
        001     /LXISP INSTRUCTION?
        JZ      /MAYBE IT IS, MAKE ANOTHER TEST.
        LXI
        0
        MOVAB   /GET THE OP CODE BACK IN A.
        ANI     /IS THE INSTRUCTION A CONDITIONAL
        301     /JUMP, CALL, OR RETURN (3X2,3X4,
        CPI     /3X0)?
        300
        JNZ     /NO, THEN IT IS A SINGLE-BYTE
        ONEBYT  /INSTRUCTION.
        0
CANTDO, MVIA    /WE CANNOT SINGLE-STEP THROUGH
        277     /THIS PARTICULAR INSTRUCTION.
        CALL    /PRINT A ? ON THE TELETYPEWRITER
        TTYOUT  /OR CRT AND THEN JUMP TO THE
        0
        JMP     /COMMAND DECODER SO THAT ANOTHER
        CMDDEC  /COMMAND CAN BE ENTERED AND
        0       /INTERPRETED.
LXI,    MOVAB   /MAYBE IT IS AN LXI-TYPE INSTRUCTION.
        ANI     /MAKE ANOTHER TEST.
        010
        JNZ     /IT IS NOT AN LXI-TYPE INSTRUCTION,
        ONEBYT  /SO IT IS A SINGLE-BYTE INSTRUCTION.
        0
THRBYT, INXH    /IT'S A THREE-BYTE INSTRUCTION.
TWOBYT, INXH    /TWO-BYTE INSTRUCTION.
ONEBYT, INXH    /ONE-BYTE INSTRUCTION.
SPCSTP, CALL    /SET A BREAKPOINT AT THE MEMORY
        SETBRK  /LOCATION ADDRESSED BY REGISTER
        0       /PAIR H (EXAMPLE 10-3).
        LHLD    /THEN LOAD REGISTER PAIR H WITH
        USERSP  /THE USER'S STACK POINTER.
        0
        SPHL    /LOAD THE SP WITH THIS ADDRESS.
        POPPSW  /POP A AND THE FLAGS OFF OF THE STACK.
        POPB    /THEN POP REGISTER PAIR B AND
        POPD    /REGISTER PAIR D OFF OF THE STACK.
        LHLD    /LOAD REGISTER PAIR H WITH THE ADDRESS
        TEMPO   /OF THE NEW BREAKPOINT.
        0
        PUSHH   /SAVE THE NEW "BRKADD" ON THE STACK.
        LHLD    /LOAD REGISTER PAIR H WITH THE
        BRKADD  /ADDRESS WHERE WE SHOULD CONTINUE
        0       /PROGRAM EXECUTION.
        XTHL    /NEW BRKADD IN H AND L, OLD BRKADD ON
        SHLD    /STACK. THEN SAVE THE NEW BRKADD
        BRKADD  /IN R/W MEMORY FOR LATER USE.
        0
        POPH    /POP THE OLD BRKADD INTO H AND L.
        XTHL    /EXCHANGE IT WITH REGISTER PAIR H.
        RET     /PUT THE OLD BRKADD IN THE PC.

DLOAD,  MOVAB   /GET THE INSTRUCTION OP CODE IN A.
        ANI     /SAVE ONLY BIT D3.
        010
        JNZ     /IT IS NOT A 042, 052, 062, OR 072
        ONEBYT  /(22, 2A, 32, OR 3A), SO IT MUST
        0       /BE A SINGLE-BYTE INSTRUCTION.
        JMP     /IT IS AN LDA, STA, LHLD, OR SHLD
        THRBYT  /INSTRUCTION, SO TREAT IT AS
        0       /SUCH.
```

At STEP in Example 10-18, the 8080 loads the A and B registers with the op code for the instruction that is stored in memory at the breakpoint address. The SETBRK subroutine saved the instruction op code in ORGIN (Example 10-3). A number of comparisons are then made between the op code for the instruction at the breakpoint address and immediate data bytes that represent the op codes for many of the 8080 instructions. These comparisons determine whether or not the instruction can be single-stepped through and, if so, the number of bytes in the instruction.

The first two comparisons compare the op code of the instruction to be single-stepped through to 323 and 333, the octal op codes for the OUT and IN instructions. If the instruction op code is equal to either of these values, the 8080 jumps to TWOBYT because OUT and IN are both two-byte instructions. If the instruction op code is not equal to either of these values, it is compared to the op code for the unconditional jump instruction, JMP. If the instruction op code is equal to this, the 8080 jumps to CANTDO, because this version of the debugger cannot single-step through the JMP instruction. In fact, if the instruction op code is equal to any of the unconditional or conditional jump, call, or return instructions, the 8080 will jump to CANTDO. Likewise, if the instruction op code is equal to the op codes for the restart or PCHL instructions, the 8080 prints a question mark on the teletypewriter or crt, and then jumps back to the command decoder contained within the debugger. This indicates to the user that the 8080 cannot single-step through this instruction.

Note that the 8080 does not have to compare the instruction op code at the breakpoint address to all of the unconditional and conditional jump, call, and return instruction op codes. This does not have to be done because there is some "consistency" in the various op codes for the 8080 instructions. For instance, bits D_7, D_6, D_2, D_1, and D_0 for all of the conditional jump instructions are the same: 1, 1, X, X, X, 0, 1, 0. By ANDing the op code for the instruction at the breakpoint address with 307, and then comparing the result to 302, the zero flag will be a logic 1 if the instruction op code is equal to any of the conditional jump instruction op codes. The 8080 can also compare this masked instruction op code to 300 to determine if the instruction is a conditional return instruction, or to 304 to determine if the instruction is a conditional call instruction. By using octal op codes for the 8080 instruction set, it is much easier to see this pattern in the instructions. In fact, the 8080 determines whether the op code represents *any* of the conditional instructions by executing an ANI 301, CPI 300 instruction sequence (careful, this is tricky).

If the 8080 can single-step through the instruction, either the instructions at THRBYT, TWOBYT, or ONEBYT are executed. Since register pair H contains the previous breakpoint address (look at the third instruction in STEP; LHLD BRKADD), these instructions increment this address the required number of times so that register pair H addresses the next "executable" instruction op code in memory. The SETBRK subroutine is then called so that the breakpoint is set at this memory location. This means that the instruction op code is saved in ORGIN, and a RST5 instruction is saved in this memory location.

After the breakpoint has been set, its address is temporarily stored (in TEMPO) so that register pair H may be used to hold the user's stack-pointer address. The user's stack contains register values for registers B, C, D, E, H, L, A, and the flags. These values are retrieved from the user's stack and are used to ready the 8080 for the "continue" sequence of instructions (command). The value for register pair H is left on the stack, since register pair H will be used again before the 8080 actually continues executing the user's program.

Various program steps are then used to store the new breakpoint address in memory locations BRKADD and BRKADD+1, to restore register pair H and to continue execution of the user's program. This is accomplished by:

1. Pushing the new breakpoint address onto the stack.
2. Loading register pair H with the old breakpoint address (the point where the 8080 *must* continue program execution).
3. Exchanging the stack and register pair H values so that the continue address is now on the stack and the new breakpoint address is in register pair H.
4. The new breakpoint address then replaces the old breakpoint address at BRKADD.
5. The continue address is placed in register pair H and then exchanged with the last stack entry. This restores register pair H to the value that it contained when the last breakpoint was reached. The continue address is now on the stack and it can be "jumped to" with the execution of the return (RET) instruction.

This is a complex portion of the program and you may wish to read through it several times. If you doubt that this sequence of instructions works, set up an imaginary stack on paper, an old breakpoint address in BRKADD, and a new breakpoint address in register pair H. Then execute, on paper, the instructions starting at SPCSTP.

When the 8080 returns to the user's program, the instruction at the old breakpoint address is executed before the RST5 instruction is encountered at the new breakpoint. When the RST5 is executed, the 8080 jumps to TRAP again, so that the contents of the registers are printed, showing the effect of "stepping" through a *single* instruction.

SINGLE-STEPPING THROUGH TRANSFER-OF-CONTROL INSTRUCTIONS

At this time, it would be far too complex to explain how a debugger can be written so that it can single-step through transfer-of-control instructions that we have already mentioned. However, it should be relatively easy for you to visualize how it might be done. If an unconditional jump or call instruction (JMP or CALL) op code is ready from memory, the second and third bytes of the instruction must be used as the address for the next breakpoint. If a RET (unconditional return) op code is read from memory, the return address of the subroutine, which is stored on the *user's* stack *above* the register pairs, must be used as the new breakpoint address. For a PCHL instruction, the content of register pair H, which is also saved on the user's stack (see the instructions at TRAP, Example 10-15), must be used as the new breakpoint address. To single-step through a restart instruction, *other than the restart instruction used for the breakpoint feature of the debugger,* the address to which the 8080 "vectors" can be easily calculated (Example 10-19).

Example 10-19: Calculating the Vector Address for a Restart Instruction

```
/THIS SECTION OF THE DEBUGGER CALCULATES THE NEW BREAK-
/POINT ADDRESS, BASED ON THE RESTART INSTRUCTION BEING
/EXECUTED.

RSTIN,  MOVAB   /MOVE THE RESTART'S OP CODE TO A.
        ANI     /SAVE ONLY BITS D5, D4, AND D3.
        070     /00XXX000.
        MOVLA   /SAVE THIS VALUE IN L.
        MVIH    /SET THE H REGISTER TO 000 (00).
        000
        JMP     /THEN SET THE BREAKPOINT AT THIS
        SPCSTP  /ADDRESS AND CONTINUE PROGRAM
        0       /EXECUTION.
```

In Example 10-19, the RSTIN section of the debugger is only executed if the octal op code 3X7 (X = don't care) is read from memory at the old breakpoint address. The 16-bit address for RSTIN would have to be written into the instruction decoder in STEP just after the CPI 307 instruction. The address for CANTDO would be changed to RSTIN (Example 10-18). The instruction op code is read from the B register into the A register, and bits D_7, D_6, D_2, D_1, and D_0 are set to 0 by the ANI instruction. If a RST3 instruction is to be single-stepped through, the op code 337 is read from memory. After the ANI instruction is executed, the A register contains 030 (18). This value is then loaded into the L register and the H register is set to 0. By jumping to SPCSTP, the

8080 sets the breakpoint at the memory location addressed by register pair H. This is the memory location that the 8080 will be "vectored to" when the RST3 instruction is executed. As you can see, by using octal op codes it is very easy to see how the 8080 determines the proper address for the next breakpoint.

The conditional jump, call, and return instructions are the most difficult instructions to single-step through. One method that can be used is to set two breakpoints. One would be set immediately after the instruction, and the other breakpoint would be set at the memory location where program control could be transferred to. If the conditional instruction is not executed, then the 8080 will reach the breakpoint stored immediately after the instruction in memory. If the conditional instruction is executed, then the 8080 will reach the breakpoint that is set in the memory location where control could possibly be transferred to. This method is complex because it means that the debugger has to set two independent breakpoints and, when one of them is reached, *both* of them must be removed from the program. We will not discuss this method any further.

The second and, perhaps, the most "elegant" method (and, we hope, the best) consists of actually "creating" a *conditional jump instruction op code from the conditional jump, call or return instruction op code.* Once the conditional instruction is converted to a conditional jump instruction, it is stored in a section of R/W memory. The 16-bit address that must also be stored in memory after this op code is the address of the instruction sequence that will be executed if the conditional jump *is* executed. After the conditional jump instruction, the debugger must also store an unconditional jump instruction in memory. The address for this instruction is the address of the sequence of instructions that will be executed if the conditional jump instruction is not executed. After the debugger "creates" the conditional jump instruction op code from the conditional instruction to be single-stepped through, it has to save it in memory, followed by a 16-bit address and an unconditional jump instruction (and its 16-bit address).

The 8080 then has to load the stack pointer with the user's stack pointer and load the PSW with the "user's" flags (POPPSW). The 8080 must do this so that the conditional jump instruction that it created actually tests the user's flags. After the 8080 executes the POPPSW instruction, it jumps to the R/W memory location that contains the conditional jump instruction that it created. The 8080 will then either execute the conditional jump instruction or the unconditional jump instruction.

If the conditional jump instruction is executed, the 8080 jumps back to a specific point in the debugger. If the 8080 gets back to this section of the debugger, then the *conditional instruction in the user's program would have been executed if we had single-stepped through it.* Therefore, the 8080 determines the address that the 8080 conditionally jumps to or calls, or the address that the 8080 conditionally returns to. A breakpoint is set at this address, and then the 8080 executes the "continue" sequence of instructions. The conditional instruction in the *user's program* is then executed and the 8080 is vectored back to the debugger.

If the conditional instruction that the 8080 created is not executed, the unconditional jump instruction that is stored after it in memory is executed. The 8080 jumps back to a section of the debugger that determines the address of the next instruction stored in memory *after* the conditional instruction in the user's program. The 8080 then sets a breakpoint at this address and executes the "continue" sequence of instructions. The conditional instruction in the user's program is not executed, so the 8080 reaches the breakpoint set immediately after it in memory and is vectored back to the debugger.

The debugger executes only three instructions to convert all of the conditional instructions to conditional jump instructions, as seen in Example 10-20. These instructions can only be executed

Example 10-20: Converting All of the Conditional Instructions to Conditional Jump Instructions

```
/THIS SECTION OF THE DEBUGGER CONVERTS ALL CONDITIONAL
/JUMP, CALL, AND RETURN INSTRUCTIONS TO CONDITIONAL
/JUMP INSTRUCTIONS.

CONVRT, MOVAB   /GET THE CONDITIONAL OP CODE IN A.
        ANI     /SAVE ONLY BITS D7, D6, D5, D4,
        370     /AND D3 (370 = F8).
        ADI     /THEN ADD 002 (02).
        002     /THE CONDITIONAL JUMP OP CODE IS
         •      /NOW IN THE A REGISTER.
         •
         •
```

because the op codes for the conditional jump, call, and return instructions are very similar. Bits D_7 and D_6 are both logic 1s for all 24 of these instructions. Bits D_5, D_4, and D_3 represent the condition that will be tested. This means that eight conditions can be tested—the true and false state of the sign, zero, carry, and parity flags. The various combinations of bits D_5, D_4, and D_3, and the conditions that they represent, can be seen in Table 10-4.

The three bits in Table 10-4 represent the same conditions, regardless of the conditional instruction (jump, call, or return). This means that bits D_5, D_4, and D_3 are the same state in the op codes for the JNZ, CNZ, and RNZ instructions. The three LSBs (D_2, D_1,

Table 10-4. The Conditions That Can Be Tested and Their Codes

Condition	Code
	D_5 D_4 D_3
NZ	0 0 0
Z	0 0 1
NC	0 1 0
C	0 1 1
PO	1 0 0
PE	1 0 1
P	1 1 0
M	1 1 1

and D_0) determine whether the instruction is a conditional jump, call, or return instruction. Table 10-5 contains the combinations of these three bits for these three classes of instructions.

As an example, let us assume that we need to single-step through a CNZ instruction. When the STEP instructions are executed, the CNZ op code (304, C4) is read into the A register. The ANI 301, CPI 300 instructions are eventually executed (Example 10-18), so the 8080 jumps to CONVRT. The op code for the CNZ instructions is then ANDed with 370, and the result, 300, is left in the A register. Two is added to this, and the result, 302 (C2), is the op code for the JNZ instruction. The 8080 stores this op code in memory, along with a 16-bit address followed by an unconditional jump instruction (JMP). The result looks like

```
JNZ
ALPHA
0
JMP
BETA
0
```

After these instructions are stored in R/W memory, the debugger loads the SP with the user's stack pointer and pops the flags off of the stack. The 8080 then jumps to the memory location containing the JNZ instruction.

If the condition tested by the JNZ instruction is met, the debugger jumps to ALPHA, where the debugger determines the appropriate

Table 10-5. The Jump, Call, and Return "Bits" in the Conditional Instruction Op Codes

Type of Instruction	D_2 D_1 D_0
Conditional Jump	0 1 0
Conditional Call	1 0 0
Conditional Return	0 0 0

address that the 8080 will transfer control to. In this case, since the debugger is stepping through a CNZ instruction, the 8080 will set a new breakpoint at the memory location addressed by the second and third bytes of the CNZ instruction.

If the debugger branches to BETA instead, the breakpoint is set at the next instruction op code stored in memory after the CNZ instruction. The 8080 then pops all of the user's registers off of the stack and jumps to the CNZ instruction stored in the program being debugged. Because of the state of the user's flags, the CNZ instruction is not executed, and the 8080 reaches the breakpoint stored in memory after the CNZ instruction.

Using this technique, the debugger determines the condition (flag) to be tested, and then whether or not the particular condition has been met. By testing the condition of the flag, *under the control of the debugger,* it can determine at which one of two possible addresses to place the next breakpoint; the address of the next sequential op code or the address to which control is to be transferred. All of the software required to perform these operations is complex, so we will not provide software listings.

A SIMPLE DEBUGGER

Now that we have discussed the "modules" or subroutines required for a simple debugger, we will link them together with a command decoder. This program will *only* be a debugger. It will not provide you with program modification or memory modification capabilities. The debugger will have five commands, which are summarized in Table 10-6.

Table 10-6. The Commands for a Simple Debugger

1. Set a breakpoint at an address.
2. Remove a breakpoint.
3. Execute a program stored in memory.
4. Single-step through an instruction
   (except a transfer-of-control instruction).
5. Continue program execution from the last breakpoint.

As we have discussed previously, the debugger will not be able to single-step through some instructions (jumps, calls, restarts, returns, and PCHL). When the TRAP section of the debugger is executed, the user's stack will be preserved. However, all of the user's registers will be saved on the user's stack. The debugger will then set up its own stack for its own use. The assembly language instructions for this debugger are listed in Example 10-21.

Example 10-21: A Simple Debugger

```
                    DW TEMPO 070 106
                    DW USERSP 070 104
                    DW STACK 070 000
                    DW ORGIN 070 100
                    DW BRKADD 070 102

          /DEBUG HAS 5 COMMANDS. IF AN "S" IS ENTERED, THE DEBUG-
          /GER WILL SINGLE-STEP. IF "C" IS ENTERED, THE 8080 WILL
          /CONTINUE PROGRAM EXECUTION AT FULL SPEED FROM THE
          /LAST BREAKPOINT. IF "K" IS ENTERED, THE BREAKPOINT WILL
          /BE REMOVED. A 16-BIT NUMBER CAN ALSO BE ENTERED, FOL-
          /LOWED BY "B" (TO SET A BREAKPOINT) OR "G" (TO BEGIN
          /EXECUTING A PROGRAM).

                    *100 000
100 000 061 DEBUG,  LXISP   /LOAD THE STACK POINTER FOR THE DE-
100 001 000         STACK   /BUGGER'S INITIAL USE.
100 002 070         0
100 003 076 IGNOR,  MVIA    /THEN TYPE A QUESTION MARK ON
100 004 077         "?"     /THE TELETYPEWRITER OR CRT.
100 005 315         CALL
100 006 104         TTYOUT
100 007 100         0
100 010 315 CMDDEC, CALL    /THEN PRINT A CARRIAGE RETURN AND
100 011 120         CRLF    /LINE FEED.
100 012 100         0
100 013 315         CALL    /THEN GET A CHARACTER FROM THE
100 014 073         TTYIN   /TELETYPEWRITER OR CRT.
100 015 100         0
100 016 376         CPI     /WAS AN "S" ENTERED?
100 017 123         "S"
100 020 312         JZ      /YES, THEN SINGLE-STEP.
100 021 011         STEP
100 022 101         0
100 023 376         CPI     /WAS A "C" ENTERED?
100 024 103         "C"
100 025 312         JZ      /YES, THEN CONTINUE PROGRAM EXECUTION
100 026 375         CONTIN  /AT FULL SPEED.
100 027 100         0
100 030 376         CPI     /WAS A "K" ENTERED?
100 031 113         "K"
100 032 312         JZ      /YES, THEN REMOVE THE BREAKPOINT
100 033 266         KILLBP  /THAT WAS NOT REACHED.
100 034 100         0
100 035 000         NOP
100 036 000         NOP
100 037 000         NOP
100 040 000         NOP
100 041 000         NOP
100 042 021         LXID    /NOT AN "S," "C," OR "K," TRY INTERPRETING
100 043 003         003     /THE ASCII VALUE AS AN OCTAL NUMBER.
100 044 000         000
100 045 315         CALL    /ENTER TWO ADDITIONAL OCTAL
100 046 140         SPCOCT  /NUMBERS.
```

```
100 047 100            0
100 050 142            MOVHD    /MOVE THE RESULT INTO H.
100 051 315            CALL     /THEN GET AN ADDITIONAL THREE
100 052 132            OCTIN    /OCTAL NUMBERS.
100 053 100            0
100 054 152            MOVLD    /MOVE THE RESULT TO L.
100 055 315            CALL     /THEN GET ANOTHER TELETYPEWRITER
100 056 073            TTYIN    /OR CRT CHARACTER.
100 057 100            0
100 060 376            CPI      /WAS A "B" ENTERED AFTER THE 6-DIGIT
100 061 102            "B"      /OCTAL NUMBER?
100 062 312            JZ       /YES, THEN SET A BREAKPOINT AT THE
100 063 233            BREAK    /MEMORY LOCATION ADDRESSED BY
100 064 100            0        /REGISTER PAIR H.
100 065 376            CPI      /WAS A "G" ENTERED AFTER THE 6-DIGIT
100 066 107            "G"      /OCTAL NUMBER?
100 067 302            JNZ      /NO, THEN GET ANOTHER COMMAND.
100 070 003            IGNOR
100 071 100            0
100 072 351   GO,      PCHL     /LOAD THE PC WITH REGISTER PAIR H.
100 073 333   TTYIN,   IN       /INPUT THE UART'S STATUS BITS.
100 074 001            001
100 075 346            ANI      /SAVE ONLY THE RECEIVER'S FLAG.
100 076 001            001      /IF A=001, A KEY IS PRESSED.
100 077 312            JZ       /IF A=000, NO KEY IS PRESSED.
100 100 073            TTYIN    /SO KEEP WAITING FOR A KEY
100 101 100            0        /TO BE PRESSED.
100 102 333            IN       /A KEY IS PRESSED, SO INPUT THE
100 103 000            000      /ASCII CHARACTER INTO THE A REGISTER.
100 104 107   TTYOUT,  MOVBA    /SAVE THE CHARACTER IN B.
100 105 333   TTYO,    IN       /INPUT THE UART'S STATUS WORD.
100 106 001            001
100 107 346            ANI      /SAVE ONLY THE TRANSMITTER'S FLAG.
100 110 004            004      /IF A=004, THE TRANSMITTER IS READY.
100 111 312            JZ       /IF A=000, THE TRANSMITTER IS BUSY.
100 112 105            TTYO     /SO KEEP WAITING FOR THE TRANSMITTER
100 113 100            0        /(PRINTER) TO FINISH, BEFORE THE
100 114 170            MOVAB    /CONTENT OF A CAN BE PRINTED.
100 115 323            OUT      /AFTER THE CHARACTER IS MOVED FROM
100 116 000            000      /B TO A, OUTPUT IT TO THE UART.
100 117 311            RET      /RETURN WITH THE CHARACTER STILL IN A.

100 120 076   CRLF,    MVIA     /LOAD THE A REGISTER WITH THE
100 121 215            215      /ASCII VALUE FOR A CARRIAGE
100 122 315            CALL     /RETURN AND THEN PRINT IT.
100 123 104            TTYOUT
100 124 100            0
100 125 076            MVIA     /THEN LOAD THE A REGISTER WITH THE
100 126 212            212      /ASCII VALUE FOR A LINE FEED AND
100 127 303            JMP      /PRINT IT.
100 130 104            TTYOUT
100 131 100            0
```

```
100 132 021   OCTIN,    LXID     /LOAD REGISTER PAIR D WITH 000 003
100 133 003             003      /(0003). E WILL CONTAIN 003 (03)
100 134 000             000      /AND D WILL CONTAIN 000 (00).
100 135 315   OCTIN1,   CALL     /GET A TELETYPEWRITER OR CRT CHARACTER
100 136 073             TTYIN    /(IT WILL BE PRINTED ON THE TELETYPE-
100 137 100             0        /WRITER OR CRT) AND RETURN WITH IT IN A.
100 140 376   SPCOCT,   CPI      /IS THE VALUE LESS THAN ASCII 0?
100 141 060             060
100 142 332             JC       /YES, THEN IGNORE THE CHARACTER.
100 143 135             OCTIN1
100 144 100             0
100 145 376             CPI      /IS IT EQUAL TO OR GREATER THAN THE
100 146 070             070      /VALUE FOR ASCII 8?
100 147 322             JNC      /YES, THEN IGNORE THE CHARACTER.
100 150 135             OCTIN1
100 151 100             0
100 152 346             ANI      /OK, IT'S ASCII 0 THROUGH 7, SET ALL
100 153 007             007      /THE BITS EXCEPT D2, D1, AND D0 TO ZERO.
100 154 107             MOVBA    /SAVE THE NUMBER TEMPORARILY IN B.
100 155 172             MOVAD    /GET THE PREVIOUS DIGITS AND ROTATE
100 156 007             RLC      /THEM TO THE LEFT THREE TIMES. THIS
100 157 007             RLC      /WILL INCREASE THEIR SIGNIFICANCE AND
100 160 007             RLC      /MAKE ROOM FOR THE NUMBER JUST ENTERED.
100 161 200             ADDB     /ADD THE NUMBER JUST ENTERED.
100 162 127             MOVDA    /SAVE THE NEW BINARY NUMBER IN D.
100 163 035             DCRE     /DECREMENT THE DIGIT COUNTER.
100 164 302             JNZ      /IF THE COUNT IS NONZERO,
100 165 135             OCTIN1   /GET ANOTHER CHARACTER.
100 166 100             0
100 167 076   SPC,      MVIA     /THEN PRINT A SPACE.
100 170 240             240
100 171 303             JMP
100 172 104             TTYOUT
100 173 100             0

100 174 117   BINOCT,   MOVCA    /SAVE THE VALUE IN C.
100 175 346             ANI
100 176 300             300
100 177 007             RLC      /ROTATE THE TWO MSB'S INTO THE LSB'S.
100 200 007             RLC
100 201 315             CALL     /NOW CALL THE BCDOUT SUBROUTINE,
100 202 226             BCDOUT   /WHICH WILL ADD 260 (B0) TO CON-
100 203 100             0        /VERT THE CONTENT OF A AND PRINT IT.
100 204 171             MOVAC    /NOW THE MIDDLE DIGIT MUST BE PRINTED.
100 205 346             ANI
100 206 070             070
100 207 017             RRC      /THEN ROTATE THEM INTO THE
100 210 017             RRC      /3 LSB'S.
100 211 017             RRC
100 212 315             CALL     /ADD 260 (B0) TO THE CONTENT
100 213 226             BCDOUT   /OF REGISTER A AND PRINT THE
100 214 100             0        /RESULT.
100 215 171             MOVAC    /NOW PRINT THE RIGHT-HAND DIGIT.
100 216 346             ANI
```

```
100 217 007            007
100 220 315            CALL
100 221 226            BCDOUT
100 222 100            0
100 223 303            JMP      /THEN PRINT A SPACE AFTER THE NUMBER.
100 224 167            SPC
100 225 100            0
100 226 306  BCDOUT,   ADI      /ADD 260 (B0) TO THE A REGISTER.
100 227 260            260
100 230 303            JMP      /PRINT THE ASCII VALUE ON THE
100 231 104            TTYOUT   /TELETYPEWRITER OR CRT.
100 232 100            0

100 233 315  BREAK,    CALL     /SET THE BREAKPOINT AT THE
100 234 254            SETBRK   /MEMORY LOCATION ADDRESSED BY REG-
100 235 100            0        /ISTER PAIR H.
100 236 076            MVIA     /NOW LOAD THE A REGISTER WITH THE
100 237 303            303      /OP CODE FOR A "JMP" INSTRUCTION.
100 240 062            STA      /STORE THIS OP CODE WHERE THE 8080
100 241 050            050      /WILL BE VECTORED TO BY THE
100 242 000            000      /"BREAKPOINT" INSTRUCTION.
100 243 041            LXIH     /THEN LOAD REGISTER PAIR H WITH THE
100 244 304            TRAP     /HI AND LO ADDRESS BYTES FOR THE
100 245 100            0        /JUMP INSTRUCTION.
100 246 042            SHLD     /THEN STORE THIS 16-BIT ADDRESS IN THE
100 247 051            051      /MEMORY LOCATIONS JUST AFTER THE
100 250 000            000      /JMP OP CODE.
100 251 303            JMP      /ONCE THE BREAKPOINT IS SET, JUMP
100 252 010            CMDDEC   /BACK TO THE COMMAND DECODER OF
100 253 100            0        /THE DEBUGGER.

100 254 176  SETBRK,   MOVAM    /GET THE "ORIGINAL" INSTRUCTION
100 255 062            STA      /AND SAVE IT IN R/W MEMORY.
100 256 100            ORGIN
100 257 070            0
100 260 066            MVIM     /THEN WRITE A RESTART INSTRUCTION OVER
100 261 357            357      /THIS INSTRUCTION IN THE PROGRAM.
100 262 042            SHLD     /SAVE THE BREAKPOINT ADDRESS
100 263 106            TEMPO    /IN "TEMPO."
100 264 070            0
100 265 311            RET      /THEN RETURN FROM SETBRK.

100 266 315  KILLBP,   CALL     /REMOVE THE BREAKPOINT FROM THE
100 267 274            CLEARB   /PROGRAM, THEN
100 270 100            0
100 271 303            JMP      /RETURN TO THE COMMAND DECODER
100 272 010            CMDDEC   /CONTAINED WITH THE DEBUGGER.
100 273 100            0

100 274 052  CLEARB,   LHLD     /LOAD REGISTER PAIR H WITH THE
100 275 106            TEMPO    /ADDRESS WHERE THE BREAKPOINT
100 276 070            0        /IS SET.
100 277 072            LDA      /THEN LOAD THE A REGISTER WITH THE
100 300 100            ORGIN    /OP CODE FOR THE INSTRUCTION THAT
```

```
100 301 070             0        /WAS AT THIS ADDRESS.
100 302 167             MOVMA    /SAVE THE OP CODE AT THE BREAKPOINT
100 303 311             RET      /ADDRESS AND THEN RETURN.

             /NOW THE "TRAP" SECTION OF THE DEBUG PROGRAM
             /ADDS THE SP TO REGISTER PAIR H. MEMORY-REFERENCE IN-
             /STRUCTIONS CAN THEN BE USED TO ACCESS THE REGISTER
             /VALUES ON THE "STACK" WITHOUT DISTURBING THE STACK.

100 304 343  TRAP,      XTHL     /H&L ON STACK, BREAKPOINT ADDR. IN H&L.
100 305 053             DCXH     /DECREMENT THE BREAKPOINT ADDRESS.
100 306 042             SHLD     /SAVE THE BREAKPOINT ADDRESS IN
100 307 102             BRKADD   /"BRKADD."
100 310 070             0
100 311 325             PUSHD    /THEN SAVE REGISTER PAIR D,
100 312 305             PUSHB    /REGISTER PAIR B,
100 313 365             PUSHPSW  /AND FINALLY A AND THE FLAGS.
100 314 041             LXIH     /NOW LOAD REGISTER PAIR H WITH
100 315 000             000      /000 000 (0000).
100 316 000             000
100 317 071             DADSP    /ADD THE SP TO REGISTER PAIR H.
100 320 042             SHLD     /SAVE THE "USER'S SP" IN R/W MEMORY
100 321 104             USERSP   /FOR LATER USE. REGISTER PAIR H
100 322 070             0        /STILL CONTAINS THE SP ADDRESS.
100 323 061             LXISP    /THEN SET A NEW SP FOR THE DEBUGGER.
100 324 000             STACK    /NONE OF THE USER'S REGISTERS CAN NOW
100 325 070             0        /BE CLOBBERED BY A RUNAWAY STACK.
100 326 315             CALL     /PRINT A CARRIAGE RETURN AND A
100 327 120             CRLF     /LINE FEED ON THE TELETYPEWRITER
100 330 100             0        /OR CRT.
100 331 176             MOVAM    /NOW READ THE FLAG WORD FROM MEMORY
100 332 315             CALL     /AND PRINT THE FLAGS AS ONES AND ZEROS.
100 333 176             BIT      /THIS IS A NEW VERSION OF THE
100 334 101             0        /"BIT" SUBROUTINE SEEN PREVIOUSLY.
100 335 043             INXH     /INCREMENT THE ADDRESS IN REGISTER PAIR H.
100 336 176             MOVAM    /GET THE "A REGISTER"
100 337 315             CALL     /AND PRINT IT OUT IN THE FORM OF
100 340 174             BINOCT   /A THREE-DIGIT OCTAL NUMBER.
100 341 100             0
100 342 026             MVID     /LOAD THE D REGISTER WITH 003
100 343 003             003      /(THREE REGISTER PAIRS TO BE PRINTED).
100 344 036  REGPR,     MVIE     /LOAD THE E REGISTER WITH THE  NUMBER
100 345 002             002      /OF REGISTERS IN A REGISTER PAIR.
100 346 043             INXH     /INCREMENT THE MEMORY ADDRESS
100 347 043             INXH     /TWICE.
100 350 176  NXTREG,    MOVAM    /GET AN EIGHT-BIT WORD FROM MEMORY
100 351 315             CALL     /AND PRINT ITS OCTAL EQUIVALENT
100 352 174             BINOCT   /ON THE TELETYPEWRITER OR CRT.
100 353 100             0
100 354 053             DCXH     /DECREMENT THE MEMORY ADDRESS
100 355 035             DCRE     /AND THE REGISTER COUNT.
100 356 302             JNZ      /IF THE COUNT IS NONZERO, PRINT
100 357 350             NXTREG   /THE CONTENT OF THE OTHER REGISTER
100 360 100             0        /IN THE REGISTER PAIR.
```

```
100 361 043            INXH    /PRINTED ONE COMPLETE REGISTER PAIR,
100 362 043            INXH    /SO INCREMENT THE MEMORY ADDRESS TWICE.
100 363 025            DCRD    /THEN DECREMENT THE REGISTER PAIR COUNT.
100 364 302            JNZ     /IF THE COUNT IS NONZERO, THEN
100 365 344            REGPR   /THE CONTENT OF ANOTHER REGISTER PAIR
100 366 100            0       /MUST BE PRINTED.
100 367 315            CALL    /ALL THE PRINTING HAS BEEN COMPLETED,
100 370 274            CLEARB  /SO REMOVE THE BREAKPOINT (THE RST5)
100 371 100            0       /FROM THE PROGRAM.
100 372 303            JMP     /GET ANOTHER COMMAND.
100 373 010            CMDDEC
100 374 100            0

            /NOW THAT THE USER'S SP HAS BEEN SAVED IN R/W MEMORY,
            /THE "CONTINUE" COMMAND MUST LOAD THE STACK POINTER
            /WITH THIS ADDRESS, BEFORE RETRIEVING THE REGISTERS
            /FROM THE STACK AND CONTINUING PROGRAM EXECUTION.

100 375 052 CONTIN,    LHLD    /LOAD REGISTER PAIR H WITH THE
100 376 104            USERSP  /USER'S SP.
100 377 070            0
101 000 371            SPHL    /THEN LOAD THE SP WITH THIS VALUE.
101 001 361            POPPSW  /POP A AND THE FLAGS OFF OF THE STACK,
101 002 301            POPB    /AND THEN POP REGISTER PAIR B AND
101 003 321            POPD    /REGISTER PAIR D.
101 004 052            LHLD    /LOAD REGISTER PAIR H WITH THE
101 005 102            BRKADD  /BREAKPOINT ADDRESS (WHERE WE
101 006 070            0       /CONTINUE FROM). EXCHANGE WITH
101 007 343            XTHL    /REGISTER PAIR H ON THE STACK.
101 010 311            RET     /POP "BRKADD" INTO THE PROGRAM COUNTER.

            /THIS SECTION OF THE DEBUGGER DETERMINES THE NUMBER
            /OF BYTES IN EVERY 8080 INSTRUCTION. HOWEVER, SOME IN-
            /STRUCTIONS CANNOT BE EXECUTED USING THE SINGLE-STEP
            /FEATURE, INCLUDING JUMP, CALL, RETURN, RESTART, AND PCHL
            /INSTRUCTIONS (ANY INSTRUCTION THAT TRANSFERS PROGRAM
            /CONTROL).

101 011 072 STEP,      LDA     /LOAD THE A REGISTER WITH THE OP CODE
101 012 100            ORGIN   /FOR THE INSTRUCTION THAT WAS STORED
101 013 070            0       /IN MEMORY AT THE BREAKPOINT ADDRESS.
101 014 107            MOVBA   /SAVE THE OP CODE IN B ALSO.
101 015 052            LHLD    /THEN LOAD REGISTER PAIR H WITH THE
101 016 102            BRKADD  /ADDRESS WHERE THE BREAKPOINT WAS
101 017 070            0       /REACHED.
101 020 376            CPI     /IS THE INSTRUCTION AN OUT?
101 021 323            323
101 022 312            JZ      /YES, THEN IT IS A TWO-BYTE
101 023 133            TWOBYT  /INSTRUCTION.
101 024 101            0
101 025 376            CPI     /IS THE INSTRUCTION AN IN?
101 026 333            333
101 027 312            JZ      /YES, THEN IT IS A TWO-BYTE
101 030 133            TWOBYT  /INSTRUCTION.
```

```
101 031 101         0
101 032 376         CPI       /IS THE INSTRUCTION A JMP?
101 033 303         303
101 034 312         JZ        /YES, THEN WE CANNOT SINGLE-STEP
101 035 003         IGNOR     /THROUGH IT. PRINT A ? ON THE
101 036 100         0         /TELETYPEWRITER OR CRT.
101 037 376         CPI       /IS THE INSTRUCTION A CALL?
101 040 315         315
101 041 312         JZ        /YES, THEN WE CANNOT SINGLE-STEP
101 042 003         IGNOR     /THROUGH IT. PRINT A ? ON THE
101 043 100         0         /TELETYPEWRITER OR CRT.
101 044 376         CPI       /IS THE INSTRUCTION A RET?
101 045 311         311
101 046 312         JZ        /YES, THEN WE CANNOT SINGLE-STEP
101 047 003         IGNOR     /THROUGH IT. PRINT A ? ON THE
101 050 100         0         /TELETYPEWRITER OR CRT.
101 051 376         CPI       /IS THE INSTRUCTION A PCHL?
101 052 351         351
101 053 312         JZ        /YES, THEN WE CANNOT SINGLE-STEP
101 054 003         IGNOR     /THROUGH IT. PRINT A ? ON THE
101 055 100         0         /TELETYPEWRITER OR CRT.
101 056 346         ANI       /SAVE ONLY BITS D7, D6, D2, D1,
101 057 307         307       /AND D0 IN THE A REGISTER.
101 060 376         CPI       /IS THE INSTRUCTION ONE OF THE RESTARTS?
101 061 307         307
101 062 312         JZ        /YES, THEN WE CANNOT SINGLE-STEP
101 063 003         IGNOR     /THROUGH IT. PRINT A ? ON THE
101 064 100         0         /TELETYPEWRITER OR CRT.
101 065 376         CPI       /IS IT AN IMMEDIATE MATH OR LOGICAL
101 066 306         306       /INSTRUCTION?
101 067 312         JZ        /YES, THEN IT IS A TWO-BYTE
101 070 133         TWOBYT    /INSTRUCTION.
101 071 101         0
101 072 376         CPI       /IS IT AN IMMEDIATE MOVE INSTRUCTION?
101 073 006         006
101 074 312         JZ        /YES, THEN IT IS A TWO-BYTE
101 075 133         TWOBYT    /INSTRUCTION.
101 076 101         0
101 077 376         CPI       /IS IT AN STA, LDA, SHLD, OR LHLD
101 100 002         002       /(DIRECT LOAD) INSTRUCTION?
101 101 312         JZ        /MAYBE IT IS, MAKE ANOTHER TEST.
101 102 165         DLOAD
101 103 101         0
101 104 376         CPI       /IS IT AN LXIB, LXID, LXIH, OR
101 105 001         001       /LXISP INSTRUCTION?
101 106 312         JZ        /MAYBE, PERFORM ANOTHER TEST ON THE
101 107 124         LXI       /INSTRUCTION OP CODE.
101 110 101         0
101 111 170         MOVAB     /GET THE OP CODE BACK IN THE A REGISTER.
101 112 346         ANI       /IS THE INSTRUCTION A CONDITIONAL JUMP,
101 113 301         301       /CALL, OR RETURN (3X2, 3X4, OR 3X0)?
101 114 376         CPI
101 115 300         300
101 116 302         JNZ       /NO, THEN THE INSTRUCTION MUST BE A
```

```
101 117 134           ONEBYT   /SINGLE-BYTE INSTRUCTION.
101 120 101           0
101 121 303           JMP      /IT IS A CONDITIONAL JUMP, CALL,
101 122 003           IGNOR    /OR RETURN, SO THE 8080 CANNOT
101 123 100           0        /SINGLE-STEP THROUGH IT.
101 124 170  LXI,     MOVAB    /MAYBE IT IS AN LXI-TYPE INSTRUCTION.
101 125 346           ANI      /MAKE ANOTHER TEST.
101 126 010           010
101 127 302           JNZ      /IT IS NOT AN LXI-TYPE INSTRUCTION,
101 130 134           ONEBYT   /SO IT IS A SINGLE-BYTE INSTRUCTION.
101 131 101           0
101 132 043  THRBYT,  INXH     /IT'S A THREE-BYTE INSTRUCTION.
101 133 043  TWOBYT,  INXH     /TWO-BYTE INSTRUCTION.
101 134 043  ONEBYT,  INXH     /ONE-BYTE INSTRUCTION.
101 135 315  SPCSTP,  CALL     /SET A BREAKPOINT AT THE MEMORY
101 136 254           SETBRK   /LOCATION ADDRESSED BY REGISTER
101 137 100           0        /PAIR H.
101 140 052           LHLD     /THEN LOAD REGISTER PAIR H WITH
101 141 104           USERSP   /THE USER'S STACK POINTER.
101 142 070           0
101 143 371           SPHL     /LOAD THE SP WITH THIS ADDRESS.
101 144 361           POPPSW   /POP A AND THE FLAGS OFF OF THE STACK.
101 145 301           POPB     /THEN POP REGISTER PAIR B AND
101 146 321           POPD     /REGISTER PAIR D OFF OF THE STACK.
101 147 052           LHLD     /LOAD REGISTER PAIR H WITH THE
101 150 106           TEMPO    /ADDRESS WHERE THE BREAKPOINT
101 151 070           0        /WAS JUST SET.
101 152 345           PUSHH    /SAVE THE NEW "BRKADD" ON THE STACK.
101 153 052           LHLD     /LOAD REGISTER PAIR H WITH THE
101 154 102           BRKADD   /ADDRESS WHERE WE SHOULD CONTINUE
101 155 070           0        /PROGRAM EXECUTION.
101 156 343           XTHL     /NEW BRKADD IN H&L, OLD BRKADD ON
101 157 042           SHLD     /STACK. THEN SAVE THE NEW BRKADD
101 160 102           BRKADD   /IN R/W MEMORY FOR LATER USE.
101 161 070           0
101 162 341           POPH     /POP THE OLD BRKADD INTO H&L.
101 163 343           XTHL     /EXCHANGE IT WITH REGISTER PAIR H.
101 164 311           RET      /PUT THE OLD BRKADD IN THE PC.

101 165 170  DLOAD,   MOVAB    /GET THE INSTRUCTION OP CODE IN A.
101 166 346           ANI      /SAVE ONLY BIT D3.
101 167 010           010
101 170 302           JNZ      /IT IS NOT A 042, 052, 062, 0R 072
101 171 134           ONEBYT   /(22, 2A, 32, OR 3A), SO IT MUST
101 172 101           0        /BE A SINGLE-BYTE INSTRUCTION.
101 173 303           JMP      /IT IS AN LDA, STA, LHLD, OR SHLD
101 174 132           THRBYT   /INSTRUCTION, SO TREAT IT AS
101 175 101           0        /SUCH.

101 176 016  BIT,     MVIC     /LOAD THE C REGISTER WITH THE NUMBER
101 177 010           010      /OF "BITS" TO BE PRINTED.
101 200 127           MOVDA    /SAVE THE NUMBER IN D.
101 201 172  NXTBIT,  MOVAD    /GET THE WORD TO BE "PRINTED."
101 202 007           RLC      /ROTATE THE MSB INTO THE CARRY,
101 203 127           MOVDA   /AND THEN SAVE THE WORD.
101 204 076           MVIA    /LOAD THE A REGISTER WITH THE
101 205 260           260     /VALUE FOR ASCII 0 (HEX B0).
101 206 322           JNC     /IF THE CARRY IS ZERO, THEN PRINT
101 207 212           ZERO    /A "0" ON THE TELETYPEWRITER OR
101 210 101           0       /CRT. IF THE CARRY IS ONE,
101 211 074           INRA    /PRINT, A "1."
101 212 315  ZERO,    CALL    /PRINT THE CONTENT OF THE A REGISTER
101 213 104           TTYOUT  /ON THE TELETYPEWRITER OR CRT.
101 214 100           0
101 215 015           DCRC    /THEN DECREMENT THE "BIT" COUNT.
101 216 302           JNZ     /IF THE COUNT IS NONZERO, TEST
101 217 201           NXTBIT  /ANOTHER BIT OF THE L REGISTER.
101 220 101           0
101 221 303           JMP     /PRINT A SPACE AFTER ALL EIGHT
101 222 167           SPC     /ONES AND ZEROS HAVE BEEN
101 223 100           0       /PRINTED.
```

The debugger listed in Example 10-21 has a number of "bugs" resulting from the fact that we have tried to keep it as short and simple as possible. When the 8080 begins to execute this debugger, what will happen if you enter the *step* or *continue* command? The 8080 will step or continue program execution, using the last breakpoint address as a starting point. This is the 16-bit address stored in BRKADD. Since we just started the debugger, we have no way of knowing what address is initially stored in BRKADD. Therefore, the step and continue commands *must only be used after a breakpoint has been used at least once.* Also, do not enter the breakpoint removal command (K) unless a breakpoint has been entered that was never reached by the debugger.

FINAL THOUGHTS ON DEBUGGERS

None of the debuggers that we have discussed can be used to debug a program that is stored in read-only memory (ROM). This is true because the 8080 cannot write a breakpoint instruction (in our examples, a RST5 instruction) into a program that is stored in ROM.

There are also a number of improvements that can be made to the debugger listed in Example 10-21. One feature would be to have the debugger print out the address where the breakpoint was reached. If this is done, it would be very easy to relate the contents of the registers to specific instructions in the program being debugged.

In most programs, register pair H is used to address memory. Therefore, when the breakpoint is reached and the contents of the registers are printed out, the content of memory addressed by reg-

Example 10-22: Using DBUG To Single-Step Through a Program
(A) The Assembled Listing of the Example Program

```
TYCHON  EDITOR-ASSEMBLER  V-2                              PAGE  01-001

                          *020 000
020 000 061   START,     LXISP    /LOAD THE STACK POINTER SO THE REG-
020 001 200              200      /ISTERS CAN BE STORED SOMEWHERE WHEN
020 002 004              004      /THE BREAKPOINT IS REACHED.
020 003 041              LXIH     /LOAD REGISTER PAIR H WITH A R/W MEMORY
020 004 100              100      /ADDRESS.
020 005 020              020
020 006 066              MVIM     /SAVE THE FOLLOWING IMMEDIATE DATA BYTE
020 007 233              233      /IN THIS MEMORY LOCATION.
020 010 076              MVIA     /LOAD THE A REGISTER WITH AN IMMEDIATE
020 011 001              001      /DATA BYTE.
020 012 053              DCXH     /DECREMENT THE CONTENT OF REGISTER PAIR H.
020 013 206              ADDM     /ADD THE CONTENT OF MEMORY TO REGISTER A.
020 014 117              MOVCA    /SAVE THIS VALUE IN THE C REGISTER.
020 015 014              INRC     /INCREMENT THE CONTENT OF THE C REGISTER.
020 016 014              INRC     /AND INCREMENT IT TWO MORE TIMES.
020 017 014              INRC
020 020 166              HLT      /THEN HALT.
```

(B) The Results Obtained by Using DBUG

```
020  000  B
020  000  G
020  000
SZ 1 P 2  A   B   C   D   E   H   L   M   S P     C S
01000110 000 000 000 000 000 000 000 024 004 200 000 135

S  020  003
01000110 000 000 000 000 000 020 100 137 004 200 000 135

S  020  006
01000110 000 000 000 000 000 020 100 233 004 200 000 135

S  020  010
01000110 001 000 000 000 000 020 100 233 004 200 000 135

S  020  012
01000110 001 000 000 000 000 020 077 375 004 200 000 135

S  020  013
10000010 376 000 000 000 000 020 077 375 004 200 000 135

S  020  014
10000010 376 000 376 000 000 020 077 375 004 200 000 135

S  020  015
10000110 376 000 377 000 000 020 077 375 004 200 000 135

S  020  016
01010110 376 000 000 000 000 020 077 375 004 200 000 135

S  020  017
00000010 376 000 001 000 000 020 077 375 004 200 000 135
```

ister pair H should also be printed. This makes it easier to debug programs that use memory for data storage. If desired, the contents of memory addressed by register pairs B and D could also be printed. The 16-bit contents of the stack pointer could also be printed, along with the last one or two 16-bit entries on the stack, regardless of whether they represent data or return addresses.

When all of this information is printed out, it would also be advantageous to print the names of the flags and registers above the contents of the flag word and the registers. Many of the features that we have mentioned in this section of the chapter have been incorporated in *DBUG: An 8080 Interpretive Debugger*[1].

A simple assembly language program and the results of using *DBUG* to single-step through the program are listed in Example 10-22.

As you already know, when the 8080 is reset, it begins program execution starting at memory address 000 000. If you have stored your system monitor and debugger starting at this address, then a jump instruction will not have to be written into memory at the address for the restart used as the breakpoint instruction. However, the jump instruction will have to be *programmed* into the ROM when the system monitor and debugger are stored in the ROM.

In all of our debugger examples, we have used the RST5 instruction as the breakpoint instruction. This means that an interrupt device cannot generate a RST5 instruction. We also mentioned in Chapter 3 that the RST0 instruction was not as general purpose as the other restart instructions, because it produced the same "effect" as a hardwired reset. For this reason, some debuggers use the RST0 instruction as the breakpoint instruction. This means that the 8080 will start program execution at 000 000 when it is reset, and also when the breakpoint instruction, RST0, is reached in the program being debugged. Therefore, the instructions that save the registers on the user's stack and then print out their contents will be printed whenever the microcomputer is reset and, of course, when the breakpoint in the program is reached.

There are many debugger features that we have not discussed. This includes multiple (simultaneous) breakpoints, debugging interrupt service subroutines in *real-time* (as they occur), and debugging a loop in a program when a "bug" appears the 341st time through the loop. There is also the problem of debugging a program that is stored in ROM. Unfortunately, these topics are beyond the scope of this book.

REFERENCES

1. Titus, C. A. and Titus, J. A. *DBUG: An 8080 Interpretive Debugger.* Howard W. Sams & Co., Inc., Indianapolis, IN, 1977.

2. Titus, C. A., Rony, P. R., Larsen, D. G., and Titus, J. A. *8080/8085 Software Design—Book 1*. Howard W. Sams & Co., Inc., Indianapolis, IN, 1978.
3. Chamberlin, H. "Debugging Aids." *Popular Electronics*, May 1977, pp 96-97.
4. Cushman, R. H. "Homegrown Debug Software Can Hone Your Design Skills." *EDN*, May 20, 1978, pp 155-159.
5. Hughes, T. P. and Sawin, D. H. III. "Breakpoint Design for Debugging Microprocessor Software." *Computer Design*, November 1978, pp 99-107.

APPENDIX A

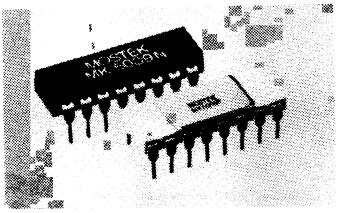
The MOSTEK MK5009 Counter Time-Base Circuit

MK 5009 P
MK 5009 N

MOS Counter Time-Base Circuit

MOSTEK

- ☐ Ion-implanted for full TTL/DTL compatibility
- ☐ Internal clock operates from:
  - External signal
  - External RC network
  - External crystal
- ☐ Operates DC to above 1 MHz
- ☐ Binary-encoded for frequency selection



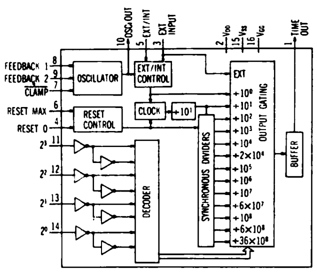
DESCRIPTION

The MK 5009 P is a highly versatile MOS oscillator and divider chain manufactured by Mostek using its depletion-load, ion-implantation process and P.-channel technology. The 16-pin DIP package provides frequency division ranges from 1 to 36x10⁸. The circuit will operate from any of three frequency sources: the internal oscillator with an external RC combination; the internal oscillator with an external crystal; or with an externally-applied TTL signal. Control inputs provide additional versatility and allow the circuit to be used in a variety of applications including instruments, timers, and clocks.

With an input frequency of 1 MHz, the MK 5009 P provides the basic time periods necessary for most frequency measuring instruments, i.e., 1 μs through 100 seconds. One-minute, ten-minute, and one-hour periods are also available using a 1 MHz input. Using a 1/1.2 MHz input, the MK 5009 P can also provide a 50/60 Hz output for accurate generation of line frequencies in portable instruments or clocks.

The time-base output (TIME OUT) is a square wave, its frequency determined by the selected counter division, and by the oscillator frequency or external input. The falling edge of the output square wave should be used to control external gating circuitry.

FUNCTIONAL DIAGRAM


10 OSC OUT
5 EXT/INT
3 EXT INPUT
2 V_{DD}
15 V_{SS}
16 V_{GG}
1 TIME OUT
FEEDBACK 1 8
FEEDBACK 2 9
CLAMP 7
RESET MAX 6
RESET 0 4
2^3 11
2^2 12
2^1 13
2^0 14
OSCILLATOR
EXT/INT CONTROL
RESET CONTROL
CLOCK
÷10¹
DECODER
SYNCHRONOUS DIVIDERS
OUTPUT GATING
EXT
÷10⁰
÷10¹
÷10²
÷10³
÷10⁴
÷2×10⁴
÷10⁵
÷10⁶
÷10⁷
÷6×10⁷
÷10⁸
÷6×10⁸
÷36×10⁸
BUFFER

TIME OUT

ADDRESS INPUTS				WITHOUT RESET	RESET		BYPASS MODES (see page 3)		
					Reset Max.	Reset Min.	Mode 1	Mode 2	Mode 3
				$R_{MAX} = 0$	$R_{MAX} = 1$	$R_{MAX} = 0$	$R_{MAX} = V_{GG}$	$R_{MAX} = 0$	$R_{MAX} = V_{GG}$
2^3	2^2	2^1	2^0	$R_O = 0$	$R_O = 0$	$R_O = 1$	$R_O = 0$	$R_O = V_{GG}$	$R_O = V_{GG}$
0	0	0	0	$\div 10^0$	$\div 10^0$	$\div 10^0$	$\div 10^0$	$\div 10^0$	$\div 10^0$
0	0	0	1	$\div 10^1$			$\div 10^1$	$\div 10^1$	$\div 10^1$
0	0	1	0	$\div 10^2$	Resets	Resets	$\div 10^2$	$\div 10^2$	$\div 10^2$
0	0	1	1	$\div 10^3$			$\div 10^3$	$\div 10^3$	$\div 10^3$
0	1	0	0	$\div 10^4$	Counters	Counters	$\div 10^4$	$\div 10^4$	$\div 10^4$
0	1	0	1	$\div 10^5$			$\div 10^2$	$\div 10^5$	$\div 10^2$
0	1	1	0	$\div 10^6$	to their	to their	$\div 10^3$	$\div 10^6$	$\div 10^3$
0	1	1	1	$\div 10^7$			$\div 10^4$	$\div 10^7$	$\div 10^4$
1	0	0	0	$\div 10^8$	Highest	Lowest	$\div 10^5$	$\div 10^5$	$\div 10^2$
1	0	0	1	$\div 6 \times 10^7$			$\div 6 \times 10^4$	$\div 6 \times 10^4$	$\div 6 \times 10^1$
1	0	1	0	$\div 36 \times 10^8$	States	States	$\div 36 \times 10^5$	$\div 36 \times 10^5$	$\div 36 \times 10^2$
1	0	1	1	$\div 6 \times 10^8$			$\div 6 \times 10^5$	$\div 6 \times 10^5$	$\div 6 \times 10^2$
	•			—			—	—	—
1	1	1	0	$\div 2 \times 10^4$			$\div 2 \times 10^1$	$\div 2 \times 10^1$	$\div 2 \times 10^1$
1	1	1	1	Ext. In.	Ext. In.	Ext. In.	Ext. Int.	Ext. Int.	Ext. Int.

*Addresses 1100 and 1101 result in Logic 0 at the output regardless of the state of the Reset Max. and Reset 0 inputs.
Logic 1 = High = V_{SS}
Logic 0 = Low = V_{DD}

Courtesy MOSTEK Corp.

ABSOLUTE MAXIMUM RATINGS

Voltage on Any Terminal Relative to V_{SS}	+ 0.3V to − 20V
Operating Temperature Range (Ambient)	0°C to +70°C
Storage Temperature Range (Ambient)	− 55°C to + 150°C

RECOMMENDED OPERATING CONDITIONS

($0°C \leq T_A \leq 70°C$)

	PARAMETER	MIN	TYP	MAX	UNITS	NOTES
V_{SS}	Supply Voltage	+ 4.5		+ 5.5	V	
V_{DD}	Supply Voltage	0.0		0.0	V	
V_{GG}	Supply Voltage	− 9.6		− 14.4	V	
f_{XTAL}	Crystal Frequency	0.1		2.0	MHz	
f_{RC}	RC Frequency	DC		200	kHz	
f_{EXT}	External Frequency	DC		2.0	MHz	
t_{PL}	Logic 0 Pulse Width, $\overline{CLAMP}$	—				Note 5
	Ext. Input	200			nsec	
t_{PH}	Logic 1 Pulse Width, Ext. Input	200			nsec	
	Reset Max	10.0			µsec	
	Reset 0	10.0			µsec	
R	Feedback Resistance	.01		2.5	MΩ	Fig. 1
V_{IL}	Input Voltage, Logic 0, Reset Inputs	0.0		0.8	V	
	Reset (Bypass Mode)	V_{GG}		V_{GG} + 1.0	V	Note 2
	All Other Logic Inputs			0.8	V	
V_{IH}	Input Voltage, Logic 1, All Logic Inputs	V_{SS}-1.0	V_{SS}	V_{SS} + 0.3	V	

ELECTRICAL CHARACTERISTICS

($V_{SS} = +5V \pm 10\%$; $V_{DD} = 0V$; $V_{GG} = -12.0V \pm 20\%$; 0 C $\leq T_A \leq$ 70 C)

	PARAMETER	MIN	TYP†	MAX	UNITS	NOTES
I_{SS}	Supply Current, V_{SS}		6.0	11.0	mA	Note 1
I_{GG}	Supply Current, V_{GG}		6.0	11.0	mA	
I_{IL}	Input Current, Logic 0			− 1.6	mA	Note 2: $V_I = 0.4V$
V_{OL}	Output Voltage, Logic 0			0.4	V	$I_{OL} = 1.6mA$*
V_{OH}	Output Voltage, Logic 1	2.4			V	$I_{OH} = -40\mu A$*
f_{STA}	Frequency Stability w/ Volt. Change, RC Mode		± 3.0		%/V	Note 3
	w/ Temp. Change, RC Mode		− 0.2		%/°C	
	Crystal Mode		—			Note 4
$t_{\bullet\bullet}$	Jitter, Edge-to-Edge Variation		<15		nsec	Temp. & Supply Voltage Constant

†Typical values at $V_{SS} = +5V$, $V_{DD} = 0V$, $V_{GG} = -12V$, and T_A 25°C

1. Logic inputs at V_{SS}, output open circuited. Each logic input (see Note 2) contributes an additional 1.6 mA (max.) to I_{SS} when at logic 0.
2. Logic Inputs are. Reset Max; Reset 0; Address Inputs; Ext. Input; Ext/Int Select; and $\overline{Clamp}$.
3. Frequency variations due to power supply changes only
4. Crystal mode stability is dependent upon crystal.
5. Minimum logic 0 time at $\overline{clamp}$ input is 50% of oscillator period

*V_{OH}, V_{OL} apply only to Time Out.

Courtesy MOSTEK Corp.

DESCRIPTION OF OPERATION

The MK 5009 P consists basically of a series of counters, selectable via an internal multiplexer. The ÷ 10^1 counter output is used to generate an internal clock signal for the 10^2 through 36×10^8 counter stages, which are fully synchronous with each other.

OSCILLATOR CONTROLS

Operation in the RC oscillator mode is achieved as shown in Figure 1. Frequency, f, is approximately 0.8/RC. The clamp circuit can be used in the RC mode to provide one-shot or accurate start-up operations. When $\overline{\text{Clamp}}$ goes to a logic 0, the internal circuitry is held at a reference level so that upon release of the $\overline{\text{Clamp}}$ (return to logic 1), the oscillator's first cycle will be a full cycle.

The crystal oscillator mode is shown in Figure 2. Values for the resistors are chosen to bias the internal circuitry for optimum performance. The two capacitors are chosen to provide the loading capacitance (C_L) specified for the selected crystal. It is recommended that $C1 = C2 = 2\ C_L$.

RESET/BYPASS CONTROLS

The MK 5009 P provides two different reset conditions. A positive-going pulse of 10 μs or longer on Reset 0 will reset counters to their lowest state, while a positive-going pulse at Reset Max will reset counters to their highest state. The Reset Max control enables the user to set up the counters to provide a falling edge at the next oscillator cycle or negative-going external input, regardless of which divider chain is selected.

In addition, taking one or both Reset Inputs to the most negative voltage, V_{GG}, allows bypassing portions of the divider chain for testing or other purposes (see table on page 1).

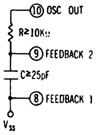
EXTERNAL/INTERNAL FREQUENCY SOURCE

When using an external signal source to operate the MK 5009 P, that signal should be applied at the External Input (Pin 3), and the External/Internal Select (Pin 5) should be brought to logic 1.

For operation with an internal signal, the External/Internal Select should be at logic 0.

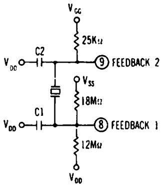
OSCILLATOR OUTPUT

The oscillator output, provided at Pin 10, is not a true logic output, but may be used to drive a high impedance device such as a junction FET or other MOS circuitry.



(10) OSC OUT
R≥10KΩ
(9) FEEDBACK 2
C≥25pF
(8) FEEDBACK 1
V_{SS}

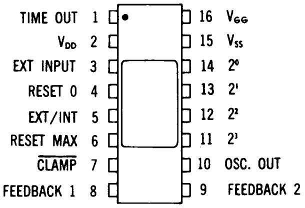
FIG. 1



V_{GG}
25KΩ
C2
V_{DD}
(9) FEEDBACK 2
V_{SS}
18MΩ
C1
V_{DD}
(8) FEEDBACK 1
12MΩ
V_{DD}

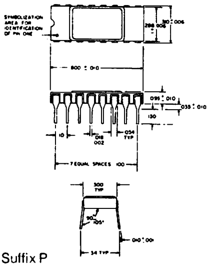
FIG. 2

PIN CONNECTIONS



TIME OUT 1
V_{DD} 2
EXT INPUT 3
RESET 0 4
EXT/INT 5
RESET MAX 6
$\overline{\text{CLAMP}}$ 7
FEEDBACK 1 8
16 V_{GG}
15 V_{SS}
14 2^0
13 2^1
12 2^2
11 2^3
10 OSC. OUT
9 FEEDBACK 2

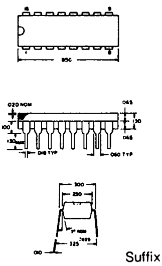
PACKAGE
16-pin ceramic dual-in-line



SYMBOLIZATION AREA FOR IDENTIFICATION OF PIN ONE
.310 ± .006
.288 .006
.800 ± .010
.095 ± .010
.035 ± .010
.130
.10
.018 .002
.054 TYP
7 EQUAL SPACES .100
.300 TYP
90° 105°
.010 ± .001
.34 TYP

Suffix P

PACKAGE
16-pin plastic dual-in-line



16
9
1
8
.850
.020 NOM
.065
.130
.100
.065
.130 MAX
.018 TYP
.060 TYP
.300
.250
.325
.010

Suffix N

Courtesy MOSTEK Corp.

APPENDIX B

Microcomputer Interfacing*

PREPARING PROGRAMS

One of the problems facing many microcomputer users is the preparation of software for particular applications. The software examples provided in past columns are short enough to have been put together or *assembled* by hand; i.e., each mnemonic was translated into its octal, hexadecimal, or binary equivalent. Addresses for jumps, calls, and input/output devices are easily added or changed since the computer programs are short and the addresses are probably listed in sequential order on the rough draft. Unfortunately, not all software preparation is this easy. Many application programs can be many thousands of steps long. This column will initiate a discussion of the aids available for microcomputer program development.

One of the biggest problems in software development is the clear concise statement of the problem and its solution. All of the desired results, inputs, outputs, and the complete program flow—including all decision-making steps—must be considered before the programming is started. This can be in outline or block diagram form, but a flowchart will prove much easier to follow. A typical flowchart is shown in Fig. B-1.

After the problem has been well thought out and a solution put in flowchart form, a decision must be made. Is the program short enough to be easily translated by hand? In many cases, particularly

---

* Reprinted from *American Laboratory*, Vol. 9, No. 10, October 1977. Copyright © 1977 by International Scientific Communications, Inc.

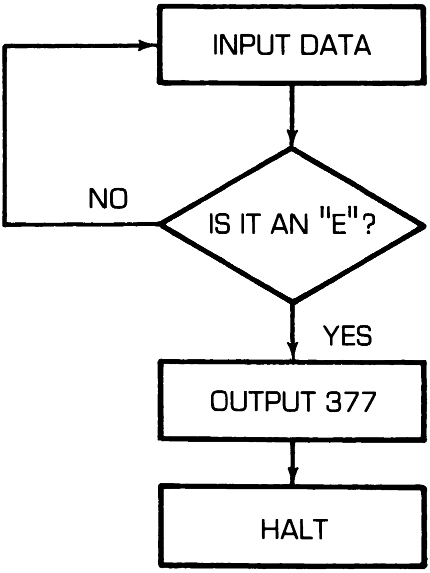


Fig. B-1. A typical flowchart.

where the programs are simple, hand assembly makes sense. In other cases software development aids called *editors* and *assemblers* are faster and more efficient. To understand how editors and assemblers work, consider the process we use to put together this column.

The first step is an outline of the subject so that we can cover it well in the short column format. A handwritten copy is then typed, corrected, retyped, and perhaps corrected and typed a final time. The illustrations and examples are formulated and drawn separately. This is the *editing* process. When writing a column, it is best to avoid references such as "the example below" or "the table on the following page." When the column is composed or *assembled*, references to a specific table or figure are much easier to follow.

Computer software is developed in much the same way. An editor program is used, either on a microcomputer or a time-sharing system, to edit the individual program steps. The editor can correct program steps, change steps, or insert and delete steps, just as a secretary would do with a manuscript. The editor program is generally unaware that you are writing a computer program, since you can use most editors to write a letter, prepare mailing lists, etc. When using an editor to prepare a program in mnemonic form, *symbolic addresses* are often assigned to software tasks within the program. In this way the actual value of the addresses for subprograms or subroutines may refer to the letters, LOOP, as the starting address of a time delay loop. Allowing us to use symbolic addresses for program steps means that the program may be changed without regard to the actual numeric values of addresses.

The assembler program must be such that it accepts information from the editor and generates an output in a form compatible with your computer. Just as you assemble short programs a step at a time, so does the assembler. The assembler contains a table of mnemonics and their equivalent values. For example, an 8080 assembler would translate an MVIA instruction into 076 octal. The assembler also assigns real, 16-bit addresses to your symbolic addresses, such as LOOP. When using symbolic addresses you must be sure to have a program step for each symbolic address, and you must assign an address if you use a symbol. You cannot assign the same "name" to more than one address. Most assemblers will recognize a *redefined symbol* or an *undefined symbol* and will produce an error message to let you know what needs to be corrected.

The final assembler output will be in punched paper tape, cassette, or disk form ready to run on your system. Most assemblers will also produce a listing of the program showing the address of each step, the data in each successive location, a symbolic address name, and the mnemonic plus any comments. A typical assembler output is shown in Example B-1.

Example B-1: Software Example Showing a Typical Assembler Output

```
                      *003 000
003 000 061  START,   LXISP    /SYMBOLIC ADDRESS OF START.
003 001 377           377
003 002 000           000
003 003 333  LOOP,    IN       /INPUT DATA FROM PORT 5.
003 004 005           005
003 005 376           CPI      /COMPARE IT TO 026.
003 006 026           026
003 007 312           JZ       /IF IT MATCHES GO TO "DETECT."
003 010 015           DETECT
003 011 003           0
003 012 303           JMP      /IF IT DOESN'T MATCH, GO TO
003 013 003           LOOP     /LOOP AND CHECK AGAIN.
003 014 003           0
003 015 171  DETECT,  MOVAC
003 016 323           OUT
003 017 007           007
003 020 166           HLT
```

After a program has been assembled, it will probably have to be debugged in order to operate properly. The program checkout and debugging can be painful without additional software "tools." Computer control panels often prove useful, but reading binary codes can become tedious, and many computers have no external controls and readouts. As an alternative, *debugging programs* are available for most microcomputers that allow you to change instructions, list blocks of data or instructions, and single-step through a program one instruction at a time.

One feature of many debug programs is the ability to establish a *breakpoint* in the software being tested. When the computer reaches a breakpoint, the instruction at that address is executed and an output device such as a teletypewriter lists the contents of important, internal CPU registers. Breakpoints are very useful since they indicate not only that the computer reached a certain point in the software, but what the computer was doing when it got there. If a breakpoint is set in the normal program flow and it is not reached, there is something wrong with the program. In this case, the breakpoint would be moved closer and closer to the start of the program until the error is found. When the error is found, it may be corrected by using the debug program to change an instruction, data, etc.

When the program is operating correctly, the debug program should have the means of saving it on paper tape, a cassette, or other medium. It should also be able to read such programs back into memory. In any case, when errors are found, you will probably want to re-edit and reassemble the software to produce a complete, error-free documented listing.

Since most programs will contain errors, it may be a good idea to have the debug program as a permanent part of your computer. It is prudent to store a debug-type program in read-only memory (ROM or PROM) since "runaway" programs being tested might alter the debug software and necessitate its having to be loaded again. There are many debug or *monitor* programs available; Intel Corporation's Insite software library lists at least four. The editor/assembler programs may also be resident in PROM and the low cost of both read/write memory and PROM "chips" suggests that many users will keep standard system programs such as editors, assemblers, and debug resident in their system. The alternative is a paper tape, cassette or disk-based software package that must be read into memory before each use.

Cross-assemblers will also generate an assembled program, but for some other computer. For example, a PDP-11 might be able to cross-assemble 8080 microcomputer programs. Cross-assemblers can be powerful programs since some incorporate simulation programs to test the program, too.

The program we use for testing programs is DBUG written by C. A. Titus, and the assembler output shown in our program examples is that produced by the Tychon Editor/Assembler (TEA).[2] Both are resident in our 8080 system on PROM chips.

TERM DEFINITIONS

Editor—A program that allows edit functions such as addition of a line or character to a program, insertion, deletion, etc. It permits you to alter your program. The input data could be anything from programs or reports to raw instrument data.

Assembler–The program that converts the assembly language code into machine code, accepting mnemonics and symbolic addresses instead of actual binary values for addresses, instructions, and data.

Monitor—A program that controls the operation of the various programs available. The monitor will be able to access the editor, assembler, or other programs.

Debugger—A program that allows the user to observe the program flow and the results of the program's operation in a step-by-step mode. A debugger may be used to change data or instructions, alter registers, etc.

Breakpoint—A special instruction that may be inserted in a program to break off the normal program control and return control to a debug-type program. When a breakpoint is executed, the debug program will indicate what the computer was doing at that point.

Cross-Assembler—An assembler program that will generate the binary code of a program for a different type of computer. For example, an 8080 cross-assembler might operate on a PDP-8 minicomputer.

REFERENCES

1. Titus, C. A. and Titus, J. A. *DBUG: An 8080 Interpretive Debugger.* Howard W. Sams & Co., Inc., Indianapolis, IN, 1977.
2. Titus, C. A. *TEA: An 8080/8085 Co-Resident Editor/Assembler.* Howard W. Sams & Co., Inc., Indianapolis, IN, 1979.

Index

A

Alphanumeric strings, sorting, 196-205
A.m/p.m. indicator, time-of-day clock, 115-119
Array, 143
ASCII strings, searching, 161-173
Assembler, 340, 343

B

Breakpoint, definition of, 343
  instructions, 287-288
  operation, 299-301
Breakpoints and debuggers, 283-333
  adding a continue command to the debugger, 305-310
  automatically setting and clearing the breakpoint, 290-295
  example debugger, 322-331
  manually setting and clearing the breakpoint, 289-290
  nondestructively printing the contents of the registers, 301-305
  saving and printing the contents of the registers, 295-299
  single-stepping, 310-322
    through transfer-of-control instructions, 318-322
Bubble sort, 188-194
  best-case/worse-case sorting times, 194
Bugs, definition of, 283

C

Circular lists, 143
Clock, real-time, 95-106
  hardware- and software-intensive, comparison of, 103-104
  hardware/software solution, 97-100
  hardware solution, 100-101
  programmable, data format for, 101-103
  time-of-day clock, 106-119
    adding an a.m/p.m. indicator, 115-119
    initializing with a fixed time, 110-115
Command decoders, 235-253
  multiple-character, 242-248
  single-letter, 235-242
    list-based, 237-242
  variable-length, 248-253
Continue command, debugger, 305-310
Cross-assembler, 342, 343

D

Data, definition of, 139
Data domain, 207
Data format, programmable real-time clock, 101-103
Data structures, 139-144
  circular lists, 143
  linear lists, 141
  linked allocation, 142-143

Data structures—cont
  node, 140-141
  sequential allocation, 142
Debugger, definition of, 343
Debuggers and breakpoints, 283-333
  adding a continue command to the debugger, 305-310
  automatically setting and clearing the breakpoint, 290-295
  breakpoint instructions, 287-288
  breakpoint operation, 299-301
  example debugger, 322-331
  manually setting and clearing the breakpoint, 289-290
  nondestructively printing the contents of the registers, 301-305
  saving and printing the contents of the registers, 295-299
  single-stepping, 310-322
    through transfer-of-control instructions, 318-322
Decoder, instruction, 313
Decoders, command, 235-253
  multiple-character, 242-248
  single-letter, 235-242
    list-based, 237-242
  variable-length, 248-253
Directory, 279
Disable interrupt instructions, 63-64
Dot-matrix look-up table, organization of, 229
Double-precision lists, 151-157

E

Editor, 340, 343
Enable interrupt instructions, 63-64
Exchange sort, 188-194
  best-case/worse-case sorting times, 194

F

Flag, interrupt, 63
Flip-flop, interrupt, 63
Front panels
  hardwired, 255-260
  software-driven, 256

H

Handshaking, 64
Hardware
  devices
    USARTs versus UARTs, 43-51
  priority interrupts, 76-84
  UART software, 17-24
Hardwired front panels, 255-260

I

Instruction decoder, 313
Instructions, breakpoint, 287-288
Insertion and exchange sorting
  comparison of, 194
  limitations of, 195
Insertion sort, 182-188
  best-case/worse-case sorting times, 188
Interfacing, microcomputer, 339-343
  preparing programs, 339-342
Interrupt(s)
  applications, 95-137
    keyboards, interrupt-driven, 119-123
      scanned, 119-123
    light-emitting-diode (LED) displays, interrupt-driven, 123-131
    real-time clock, 95-106
      hardware- and software-intensive, comparison of, 103-104
      hardware/software solution, 97-100
      hardware solution, 100-101
      programmable, data format for, 101-103
    time-of-day clock, 106-119
      adding an a.m./p.m. indicator, 115-119
      initializing with a fixed time, 110-115
  definition of, 59
  8085, 84-88
  flag, 63
  flip-flop, 63
  hardware priority, 76-84
  instructions, 62-63
    enable and disable, 63-64

Interrupt(s)—cont
  mask, 85
  multilevel, 60
  operation, 59-60
  polled, 65-66, 70-75
  polling, 55-56
  priority, 59-60, 75-76
    software timing, 88-94
  priority interrupt control unit (PICU), 131-137
  response time, 60
  single-line, 60, 65-66
  vectored, 60, 66-75

K

Keyboards, interrupt-driven, 119
  scanned, 119-123

L

Light-emitting-diode (LED) displays, interrupt-driven, 123-131
Linked allocation, 142-143
List(s)
  double-precision, 151-157
  linear, 141
  single-precision, 145-151
  terminator, 162-163
  triple precision, 157-159
Look-up tables, 207-233
  applications of, 208
  dot-matrix, organization of, 229
  paper-tape letterer, 232
  sine, 213
  16-bit sine, 221

M

Mask, interrupt, 85
Memory-mapped UARTs and USARTs, 51-54
Menu, 279
Microcomputer interfacing, 339-343
  preparing programs, 339-342
Monitor, 342, 343
Monitors, system, 255-281
  for multiplexed display, 273-278
  for non-ASCII keyboards, 271-278
  general characteristics, 260-261
Monitors, system—cont
  hardwired front panels, 255-260
  linking programs with, 278-281
  simple program, 262-271
  software-driven front panels, 256
Multilevel interrupt, 60
Multiple-character command decoders, 242-248
Multiplexed display, system monitor for, 273-278
Multiplexed LED displays, interrupt-driven, 123-131

N

Node, 140-141
Non-ASCII keyboards, system monitor for, 271-278
Null link, 143
Numeric values
  sorting, 182-195
    exchange, 188-194
    insertion, 182-188

O

Operation, breakpoint, 299-301

P

Paper-tape letterer
  look-up table, 232
  program, 224-233
Polled interrupts, 65-66, 70-75
Polling, 55-56
Priority, 59-60
  interrupt(s), 75-76
    control unit (PICU), 131-137
    software timing, 88-94
Programmable real-time clock, data format for, 101-103

R

Real-time clock, 95-106
  hardware- and software-intensive, comparison of, 103-104
  hardware/software solution, 97-100
  hardware solution, 100-101

Real-time clock—cont
  programmable, data format for, 101-103
  time-of-day clock, 106-119
    adding an a.m./p.m. indicator, 115-119
    initializing with a fixed time, 110-115
Registers
  nondestructively printing the contents of, 301-305
  saving and printing the contents of, 295-299
Response time, 60

S

Saving and printing the contents of the registers, 295-299
Scanned keyboards, interrupt-driven, 119-123
Searching, 145-180
  ASCII strings, 161-173
  common subroutine characteristics, 159-161
  double-precision lists, 151-157
  single-precision lists, 145-151
  triple-precision lists, 157-159
Sentinel, 142
Sequential allocation, 142
Serial data format, 16-17
Setting and clearing the breakpoint
  automatically, 290-295
  manually, 289-290
Sine look-up table, 213
  16-bit, 221
Single-letter command decoders, 235-242
  list-based, 237-242
Single-line interrupt, 60, 65-66
Single-precision lists, 145-151
Single-stepping, 310-322
  through transfer-of-control instructions, 318-322
Software-based UARTs, 24-36
Software-driven front panels, 256
Sort, 143
Sorting, 181-205
  alphanumeric strings, 196-205
Sorting—cont
  numeric values, 182-195
    exchange sort, 188-194
    insertion sort, 182-188
String, 143
Symbol manipulation, 143
System monitors, 255-281
  for multiplexed display, 273-278
  for non-ASCII keyboards, 271-278
  general characteristics, 260-261
  hardwired front panels, 255-260
  linking programs with, 278-281
  simple program, 262-271
  software-driven front panels, 256

T

Tables, look-up, 207-233
  applications of, 208
  dot-matrix, organization of, 229
  paper-tape letterer, 232
  sine, 213
  16-bit sine, 221
Terminator
  list, 162-163
  table, 143
Time-of-day clock, 106-119
  adding an a.m./p.m. indicator, 115-119
  initializing with a fixed time, 110-115
Triple-precision lists, 157-159

U

Universal Asynchronous Receiver/Transmitter (UART), 13-24, 43-54
  features, 15-16
  software-based, 24-36
Universal Synchronous/Asynchronous Receiver/Transmitter (USART), 15, 43-54

V

Variable-length command decoders, 248-253
Vectored interrupt, 60, 66-75

TO THE READER

This book is one of an expanding series of books that will cover the field of basic electronics and digital electronics from basic gates and flip-flops through microcomputers and digital telecommunications. We are attempting to develop a mailing list of individuals who would like to receive information on the series. We would be delighted to add your name to it if you would fill in the information below and mail this sheet to us. Thanks.

1. I have the following books:

____

____

____

____

____

____

____

____

____

____

____

____

2. My occupation is: ☐ student ☐ teacher, instructor ☐ hobbyist

☐ housewife ☐ scientist, engineer, doctor, etc. ☐ businessman

☐ Other: ______________________

Name (print): ______________________

Address ______________________

City ______________________ State ______________

Zip Code ______________________

Mail to:

Books
P.O. Box 715
Blacksburg, Virginia 24060

The Blacksburg Group

According to Business Week magazine (Technology July 6, 1976) large scale integrated circuits or LSI "chips" are creating a second industrial revolution that will quickly involve us all. The speed of the developments in this area is breathtaking and it becomes more and more difficult to keep up with the rapid advances that are being made. It is also becoming difficult for newcomers to "get on board."

It has been our objective, as The Blacksburg Group, to develop timely and effective educational materials and aids that will permit students, engineers, scientists and others to quickly learn how to apply new technologies to their particular needs. We are doing this through a number of means, textbooks, short courses, monthly computer interfacing columns and through the development of educational "hardware" or training aids.

Our group members make their home in Blacksburg, found in the Appalachian Mountains of southwestern Virginia. While we didn't actively start our group collaboration until the Spring of 1974, members of our group have been involved in digital electronics, minicomputers and microcomputers for some time.

Some of our past experiences and on-going efforts include the following:

—The design and development of the Mark-8 computer, featured in Radio-Electronics magazine in July 1974. This is generally recognized as the first widely available hobby computer. It was based upon the 8008 processor chip. Since then we have designed the Micro-Designer (MD-1) and the Mini-Micro Designer (MMD-1). This last computer was also featured in Radio-Electronics as the Dyna-micro.

—The Blacksburg Continuing Education Series™ covers subjects ranging from basic electronics through microcomputers, operational amplifiers, and active filters. Test experiments and examples have been provided in each book. We are strong believers in the use of detailed experiments and examples to reinforce basic concepts. This series originally started as our Bugbook series and many titles are now being translated into Chinese, Japanese, German and Italian.

—We have pioneered the use of small, self-contained computers in hands-on courses aimed at microcomputer users. Our expanding line of solderless breadboarding modules or OUTBOARDS® make the design and testing of circuits much easier than was possible in the past. Our educational hardware is marketed by E & L Instruments, Inc., Derby, CT 06418, USA.

—Our short course programs have been presented throughout the world. Programs are offered through Tychon Incorporated and Virginia Polytechnic Institute and State University Extension Division. Each provides hands-on experiences with digital electronics and microcomputer hardware and software. Continuing Education Units (CEUs) are provided. Courses are presented to open groups, companies, schools and other groups. We are strong believers in hands-on experience in these courses, so much time is spent in laboratory sessions.

For additional information about the short course programs, we encourage you to write or call Dr. Linda Leffel, Continuing Education Center, VPI & SU, Blacksburg, VA 24061. Phone (703) 961-5241, or Dr. Chris Titus at Tychon, Inc., Blacksburg, VA 24060. Phone (703) 951-9030.

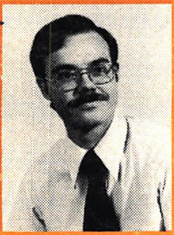
Mr. David Larsen is on the faculty of the Department of Chemistry at Virginia Polytechnic Institute and State University. Dr. Jonathan Titus and Dr. Christopher Titus are with Tychon, Inc., all of Blacksburg, Virginia.

Bugbook and OUTBOARDS are registered trademarks of E & L Instruments, Inc., Derby, CT 06418

8080/8085 Software Design Book 2

Although many microcomputers are programmed in the BASIC language, assembly language programming will continue to be used for many years to come due to the inherent limitations of most BASIC interpreters. In fact, assembly language programs can probably solve more problems than BASIC programs can! For this reason, this second volume has been added to the *8080/8085 Software Design* series. Book 1 of this series is an excellent introduction to assembly language programming and provides a good reference for the basic 8080 instruction set.

Subjects covered in this volume include asynchronous serial communications, interrupts and their applications, data structures, searching, sorting, look-up tables, command decoders, system monitors, breakpoints, and debuggers. In addition, there are over 90 executable program examples presented in the book that can be run on just about any 8080-based microcomputer.



Dr. Christopher A. Titus is a microcomputer applications engineer with Tychon, Inc., in Blacksburg, Virginia. He received his Ph.D. from Virginia Polytechnic Institute while working on microcomputer automated chemical instruments. He has coauthored a number of instrumentation articles and has had papers presented at major engineering and science conferences.

Chris has programmed with the Intel 8008, Intel 8080, and the MOS Technology 6502 microcomputers. He has written editor, assembler, disassembler, and debug software, as well as complete operating systems for microcomputers. He is also a proficient PDP-8 programmer and digital designer.

David G. Larsen is an instructor in the Department of Chemistry at Virginia Polytechnic Institute & State University, where he teaches undergraduate and graduate courses in analog and digital electronics. He is coauthor of other books in the *Blacksburg Continuing Education Series*™ and the monthly columns on microcomputer interfacing. He is a coinstructor, along with Dr. Rony, of a series of one- to five-day workshops on the digital and microcomputer revolution, taught under the auspices of the Extension Division of the University, which attract professionals from all parts of the world.

Dr. Jonathan A. Titus is the president of Tychon, Inc., in Blacksburg, Virginia. Most of his current work involves technical writing and the application of microcomputers for data acquisition and control. He has written and coauthored a number of articles on computers for both professional and popular applications.

Jon's first microcomputer experience was with the 8008, and his Mark-8 computer was featured as the first widely available hobby computer. His interests now center around the 8080 and 16-bit microcomputers. He has coinstructed courses with the American Chemical Society and now works with the Tychon hardware and software programs.

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

$9.95/21615

ISBN: 0-672-21615-9